A Block-Asynchronous Relaxation Method for Graphics Processing Units

H. Anzt, J. Dongarra, V. Heuveline, S. Tomov

No. 2011-14

Preprint Series of the Engineering Mathematics and Computing Lab (EMCL)

$\frac{\partial u}{\partial t} = \nu \Delta u$
$+ (u \cdot \nabla) u + \frac{1}{\rho} \nabla p$
$\frac{\partial u}{\partial t}$

Preprint Series of the Engineering Mathematics and Computing Lab (EMCL)
ISSN 2191–0693
No. 2011-14

Impressum

Karlsruhe Institute of Technology (KIT)
Engineering Mathematics and Computing Lab (EMCL)

Fritz-Erler-Str. 23, building 01.86
76133 Karlsruhe
Germany

KIT – University of the State of Baden Wuerttemberg and
National Laboratory of the Helmholtz Association

A Block-Asynchronous Relaxation Method for Graphics Processing Units

Hartwig Anzt*, Stanimire Tomov†, Jack Dongarra†‡§ and Vincent Heuveline*
* *Karlsruhe Institute of Technology, Germany*
† *University of Tennessee Knoxville, USA*
‡ *Oak Ridge National Laboratory, USA*
§ *University of Manchester, UK*
{*hartwig.anzt, vincent.heuveline*}*@kit.edu*
{*tomov, dongarra*}*@cs.utk.edu*

***Abstract*—In this paper, we analyze the potential of asynchronous relaxation methods on Graphics Processing Units (GPUs). For this purpose, we developed a set of asynchronous iteration algorithms in CUDA and compared them with a parallel implementation of synchronous relaxation methods on CPU-based systems. For a set of test matrices taken from the University of Florida Matrix Collection we monitor the convergence behavior, the average iteration time and the total time-to-solution time. Analyzing the results, we observe that even for our most basic asynchronous relaxation scheme, despite its lower convergence rate compared to the Gauss-Seidel relaxation (that we expected), the asynchronous iteration running on GPUs is still able to provide solution approximations of certain accuracy in considerably shorter time than Gauss-Seidel running on CPUs. Hence, it overcompensates for the slower convergence by exploiting the scalability and the good fit of the asynchronous schemes for the highly parallel GPU architectures. Further, enhancing the most basic asynchronous approach with hybrid schemes – using multiple iterations within the ”subdomain” handled by a GPU thread block and Jacobi-like asynchronous updates across the ”boundaries”, subject to tuning various parameters – we manage to not only recover the loss of global convergence but often accelerate convergence of up to two times (compared to the standard but difficult to parallelize Gauss-Seidel type of schemes), while keeping the execution time of a global iteration practically the same. This shows the high potential of the asynchronous methods not only as a stand alone numerical solver for linear systems of equations fulfilling certain convergence conditions but more importantly as a smoother in multigrid methods. Due to the explosion of parallelism in todays architecture designs, the significance and the need for asynchronous methods, as the ones described in this work, is expected to grow.**

***Keywords*-Asynchronous Relaxation; Chaotic Iteration; Graphics Processing Units (GPUs); Jacobi Method;**

I. Introduction

The latest developments in hardware architectures show an enormous increase in the number of processing units (computing cores) that form one processor. The reason for this varies from various physical limitations to energy minimization considerations that are at odds with further scaling up of processor' frequencies – the basic acceleration method used in the architecture designs for the last decades [1]. Only by merging multiple processing units into one processor does further acceleration seem possible. One example where this core gathering is carried to extremes is the GPU. The current high-end products of the leading GPU providers consist of 448 CUDA cores for the NVIDIA Fermi generation [2] and 3072 stream processors for the Northern Islands generation from ATI [3]. While the original purpose of GPUs was graphics processing, their enormous computing power also suggests the usage as accelerators when performing parallel computations. Yet, the design and characteristics of these devices pose some challenges for their efficient use. In particular, since the synchronization between the individual processing units usually triggers considerable overhead, it is attractive to employ algorithms that have a high degree of parallelism and only very few synchronization points.

On the other hand, numerical algorithms usually require this synchronization. For example, when solving linear systems of equations with iterative methods like the Conjugate Gradient or GMRES, the parallelism is usually limited to the matrix-vector and the vector-vector operations (with synchronization required between them) [4] [5] [6]. Also, methods that are based on component-wise updates like Jacobi or Gauss-Seidel have synchronization between the iteration steps [7] [8]: no component is updated twice (or more) before all other components are updated. Still, it is possible to ignore these synchronization steps, which will result in a chaotic or asynchronous iteration process. Despite the fact that the numerical robustness and convergence properties severely suffer from this chaotic behavior, they may be interesting for specific applications, since the absence of synchronization points make them perfect candidates for highly parallel hardware platforms. The result is a tradoff: while the algorithm's convergence may suffer from the asynchronism, the performance can benefit from the superior scalability.

In this paper, we want to analyze the potential of employing asynchronous iteration methods on GPUs by analyzing convergence behavior and time-to-solution when iteratively solving linear systems of equations. We split this paper into the following parts: First, we will shortly recall the mathematical idea of the Jacobi iteration method and derive the component wise iteration algorithm. Then the idea of an asynchronous relaxation method is derived, and some basic characteristics concerning the convergence demands are summarized. The section about the experiment framework will first provide information about the linear systems of equations we target. The matrices affiliated with the systems are taken from the University of Florida matrix collection. Then we describe the asynchronous iteration method for GPUs that we designed. In the following section we analyze the experiment results with focus on the convergence behavior and the iteration times for the different matrix systems. In section V we summarize the results and provide an outlook about future work in this field.

II. Mathematical Background

A. Jacobi Method

The Jacobi method is an iterative algorithm for finding the approximate solution for a linear system of equations

$$Ax = b, \tag{1}$$

where A is strictly or irreducibly diagonally dominant. One can rewrite the system as $(L + D + U)x = b$ where D denotes the diagonal entries of A while L and U denote the lower and upper triangular part of A, respectively. Using the form $Dx = b - (L + U)x$, the Jacobi method is derived as an iterative scheme

$$x^{m+1} = D^{-1}(b - (L + U)x^m).$$

Denoting the error at iteration $m+1$ by $e_{m+1} \equiv x^{m+1} - x$, this scheme can also be rewritten as $e_{m+1} = (I - D^{-1}A)e_m$. The matrix $M \equiv I - D^{-1}A$ is often referred to as *iteration matrix*. The Jacobi method provides a sequence of solution approximations with increasing accuracy when the spectral radius of the iteration matrix M is less than one (i.e., $\rho(M) < 1$) [9].

The Jacobi method can also be rewritten in the following component-wise form:

$$x_i^{m+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^m \right). \tag{2}$$

B. Asynchronous Iteration Methods

For computing the next iteration in a relaxation method, one usually requires the latest values of all components. For some algorithms, e.g. Gauss-Seidel [7], 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 no component can be updated several times before all the other components are updated.

The question of interest that we want to investigate is, what happens if this order is not adhered. Since in this case, the individual components are updated independently and without consideration of the current state of the other components, the resulting algorithm is called chaotic or asynchronous iteration method. Back in the 70's Chazan and Miranker analyzed some basic properties of these methods, and established convergence theory [10]. In the last 30 years, these algorithms were subject of dedicated research activities [11], [12] [13] [14] [15] [16]. However, they did not play a significant role in high-performance computing, due to the superior convergence properties of synchronized iteration methods. Today, due to the complexity of heterogeneous hardware platforms and the high number of computing units in parallel devices like GPUs, these schemes may become interesting again: they do not require explicit synchronization between the computing cores, probably even located in distinct hardware devices. Since the synchronization usually thwarts the overall performance, it may be true that the asynchronous iteration schemes overcompensate the inferior convergence behavior by superior scalability.

The chaotic-, or asynchronous-relaxation scheme defined by Chazan and Miranker [10] can be characterized by two functions, an update function $u(\cdot)$ and a shift function $s(\cdot, \cdot)$. For each non-negative integer ν, the component of the solution approximation x that is updated at step ν is given by $u(\nu)$. For the update at step ν, the m^{th} component used in this step is $s(\nu, m)$ steps back. All the other components are kept. This can be expressed as:

$$x_l^{\nu+1} = \begin{cases} \sum_{m=1}^{N} b_{l,m} x_m^{\nu - s(\nu,m)} + d_l & \text{if } l = u(\nu) \\ x_l^\nu & \text{if } l \neq u(\nu). \end{cases} \tag{3}$$

Furthermore, the following conditions can be defined to guarantee the well-posedness of the algorithm [17]:

1) The update function $u(\cdot)$ takes each of the values l for $1 \le l \le N$ infinitely often.
2) The shift function $s(\cdot, \cdot)$ is bounded by some $\bar{s}$ such that $0 \le s(\nu, m) \le \bar{s}\ \forall \nu \in \{1, 2, \dots\}, \forall m \in \{1, 2, \dots, N\}$. For the initial step, we additionally require $s(\nu, m) \le \nu$.
3) The shift function $s(\cdot, \cdot)$ is independent of m.

If these conditions are satisfied and $\rho(|M|) < 1$ (i.e., the spectral radius of the iteration matrix, taking the absolute values for its elements, to be smaller than one), the convergence of the asynchronous method is fulfilled [17].

Depending on the exchange of the updated components, Baudet classified the asynchronous iterative methods into three sub-methods [18]:

1) The purely asynchronous method (PA);
2) The asynchronous Jacobi method (AJ);
3) The asynchronous Gauss-Seidel method (AGS).

The PA method releases each new value immediately after its computation, while the AJ and AGS methods exchange new values only at the end of each iteration. The only difference between the AJ and AGS methods is the choice of the values of unknowns within each iteration. The AGS method uses new values of unknowns in its subsequent

method	broadcast	used values	bound for shift
PA	Immediately	Latest available	$\lvert s(\nu, m)\rvert < \bar{s}$
AJ	End of Iter.	Begin of Iter.	$0 \leq s(\nu, m) < \bar{s}$
AGS	End of Iter.	Latest available	$-1 \leq s(\nu, m) < \bar{s}$

Table I: Basic properties of the different subclasses of asynchronous iteration methods.

updates as soon as they are computed in the same iteration, while the AJ method uses only values that are set at the beginning of an iteration. In general, the term asynchronous iteration method that we use refers to the PA method. The basic properties are summarized in Table I.

Since the barrier synchronization between the iterations is usually daunting when using highly parallel devices, the purely asynchronous method is most suitable for both communication- and synchronization-avoiding iterative implementations.

The GPU implementation of the asynchronous iteration method that we consider in III-C is of purely asynchronous nature. For convenience, from now on we will use the term asynchronous iteration method if we refer to the PA iteration.

III. Experiment Framework

A. Linear Systems of Equations

In our experiments, we search for the approximate solutions of linear system 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 considered the experiment matrices have to be properly chosen. While for the Jacobi method a sufficient condition for convergence is clearly $\rho(M) = \rho(I - D^{-1}A) < 1$ (i.e., the spectral radius of the iteration matrix M to be smaller than one), the convergence theory for asynchronous iteration methods is more involved (and is not the subject of this paper). In [17] John C. Strikwerda has shown, that a sufficient condition for the asynchronous iteration to converge for all update and shift functions satisfying conditions (1), (2) and (3) in II-B is the condition $\rho(|M|) < 1$, where $|M|$ is derived from M by replacing its elements by their corresponding absolute values.
Due to these considerations, we choose to only analyze symmetric, positive definite systems, where the Jacobi method converges. The matrices and their descriptions are summarized in Table II, their structures can be found in Figure 1. Table III additionally provides some of the convergence related characteristics of the test matrices as well as of their corresponding iteration matrices.

We furthermore take the number of right-hand sides to be one for all linear systems.

B. Hardware and Software Issues

The experiments were conducted on a heterogeneous GPU-accelerated multicore system located at the University

Matrix name	Description	$\#n$	$\#nnz$
Chem97ZtZ	statistical problem	2,541	7,361
fv1	2D/3D problem	9,604	85,264
fv2	2D/3D problem	9,801	87,025
fv3	2D/3D problem	9,801	87,025
s1rmt3m1	structural problem	5,489	262,411
Trefethen_2000	combinatorial problem	2,000	41,906

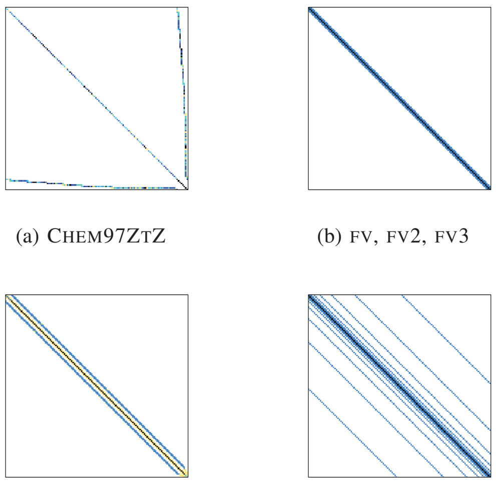
Table II: Dimension and characteristics of the SPD test matrices.

Matrix name	cond(A)	cond($D^{-1}A$)	$\rho(M)$
Chem97ZtZ	1.3e+03	7.2e+03	0.7889
fv1	9.3e+04	12.76	0.8541
fv2	9.5e+04	12.76	0.8541
fv3	3.6e+07	4.4e+03	0.9993
s1rmt3m1	2.2e+06	7.2e+06	2.65
Trefethen_2000	5.1e+04	6.1579	0.8601

Table III: Convergence characteristics of the test matrices and of their corresponding iteration matrices.

of Tennessee, Knoxville. The system's CPU is one socket Intel Core Quad Q9300 @ 2.50GHz and the GPU is a Fermi C2050 (14 Multiprocessors x 32 CUDA cores @1.15GHz, 3 GB memory). The GPU is connected to the CPU host through a PCI-e$\times$16.

In the synchronous implementation of Gauss-Seidel on the CPU, 4 cores are used for the matrix-vector operations that can be parallelized. Intel compiler 11.1.069 [19] is used with optimization flag "-O3". The GPU implementations of the asynchronous iteration and the Jacobi method are based on CUDA [20], while the respective libraries used are from CUDA 4.0.17 [21]. The component updates were coded in CUDA, using thread blocks of size 512. The kernels are then


(a) Chem97ZtZ (b) fv, fv2, fv3

(c) s1rmt3m1 (d) Trefethen_2000

Figure 1: Sparsity plots of test matrices.

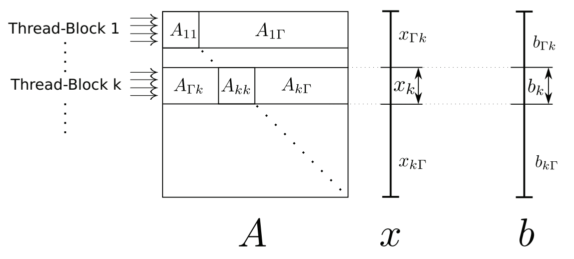
launched through different streams. The thread block size, the number of streams, along with other parameters, were determined through empirically based tuning.

C. An asynchronous iteration method for GPUs

The asynchronous iteration method for GPUs that we propose is split into two levels. This is due to the design of graphics processing units and the CUDA programming language.
The linear system of equations is split into blocks of rows, and the computations for each block is assigned to one thread block on the GPU. For these thread blocks, a PA iteration method is used, while on each thread block, a Jacobi-like iteration method is performed. We denote this algorithm by *async-(1)*.
Further, we extended this basic algorithm to a version where the threads in a thread block perform multiple Jacobi iterations (e.g., 5) within the block. During the local iterations the x values used from outside the block are kept constant (equal to their values at the beginning of the local iterations). After the local iterations, the updated values are communicated. This approach was also analyzed in [22] and is inspired by the well know hybrid relaxation schemes [23] [24], and therefore we denote it as a *block-asynchronous approach*. In other words, using domain-decomposition terminology, our blocks would correspond to subdomains and thus we additionally iterate locally on every subdomain. We denote this scheme by *async-(i)*, where the index i indicates that we use i Jacobi updates on the subdomain. Another motivation for this comes from the hardware side, especially the fact that the additional iterations almost come for free (as the subdomains are relatively small and the data needed largely fits into the multiprocessor's cache). The obtained algorithm, visualized in Figure 2, can be written as component-wise update of the solution approximation:

$$x_k^{(m+1)} += \frac{1}{a_{kk}} \left(\underbrace{b_k - \sum_{j=1}^{T_S} a_{kj} x_j^{(m-\nu(m+1,j))}}_{\text{global part}} \underbrace{- \sum_{j=T_S}^{T_E} a_{kj} x_j^{(m)}}_{\text{local part}} \underbrace{- \sum_{j=T_E}^{n} a_{kj} x_j^{(m-\nu(m+1,j))}}_{\text{global part}} \right), \tag{4}$$

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 always antecedent values are used, while for the global part, the values from the beginning of the iteration are used. The shift function $\nu(m+1,j)$ denotes the iteration shift for the component j - this can be positive or negative, depending on whether the respective other thread block already has 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.



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

Figure 2: Visualizing the asynchronous iteration in block description used for the GPU implementation.

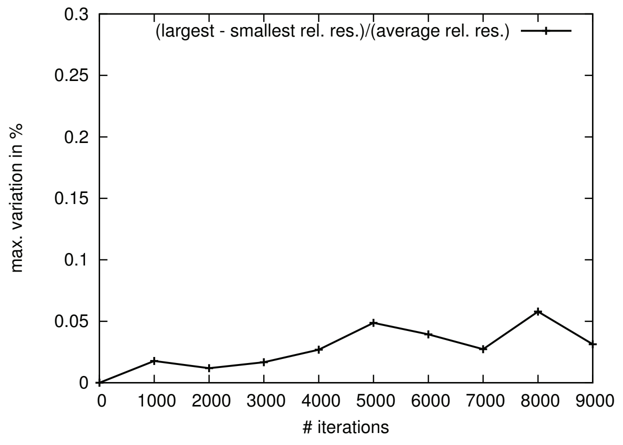
IV. Numerical Experiments

A. Stochastic impact of chaotic behavior of asynchronous iteration methods

At this point it should be mentioned, that only the synchronous Gauss-Seidel and Jacobi methods are deterministic. For the asynchronous iteration method on the GPU, the results are not reproducible at all, since for every iteration run, a very unique pattern of component updates is conducted. It may be possible, that another component update order may result in faster or slower convergence. Especially when the component update order of the synchronous Jacobi or Gauss-Seidel method is chosen, considerable performance increase can be expected.

The variations in the convergence behavior will increase with the number of iterations, since for higher iteration numbers, the random component updates multiply their influence. Additionally, the influence of the unique component update order may have even more impact, when iterating locally. This was also observed in our experiments. Also the matrix properties like condition number may have some impact.

Figure 3: Statistic convergence behaviour of asynchronous iteration method.



But the important question is, whether these stochastic effects are crucial when using asynchronous iteration methods.

To investigate this issue, we conduct multiple solver runs using the same experimental setup and monitor the relative residual behavior for the different simulations. To enhance the effect, we use the async-(5) algorithm and merge multiple kernels into one stream.

The results reported in Table IV are based on 100 simulation runs on the test matrix FV3. Additionally we provide in IV information about the statistical parameter like variance, standard deviation and standard error. Analyzing the results, we observe only small variations in the convergence behavior: for 1000 iterations the relative residual is improved by $5 \cdot 10^{-2}$, the maximal variation between the fastest and the slowest convergence rate is in the order of 10^{-5}. While the order of the residual decreases for additional iterations by a factor of 10^{-1} for 1000 iterations, this is almost also true for the maximal variation of the convergence rate. Achieving an average relative residual of $1.438192E-11$ after 9000 iterations, the variation relative residual difference of the best and worst converging solver run is in the order of 10^{-15}.

Analyzing Figure 3, we observe that the relative variation of the fastest and slowest convergence shows only a very small increase with respect to the iteration number.

Since in all experiments we never observed variations in the convergence rate of the asynchronous iteration method larger than 1%, it seems reasonable to neglect the issue of the unique component update pattern, and to state the further results for the asynchronous iteration method as average, knowing that a different solution update pattern may lead to a slightly faster or slower convergence rate.

B. *Convergence rate of the asynchronous iteration method*

In the next experiment, we analyze the convergence behavior of the asynchronous iteration method and compare it with the convergence rate of the Gauss-Seidel and Jacobi method.

# iters	variance $\left(\frac{\sum_i(x_i-\bar{x})}{n-1}\right)$	standard deviation $\left(\sqrt{\frac{\sum_i(x_i-\bar{x})}{n-1}}\right)$	standard error $\left(\sqrt{\frac{\sum_i(x_i-\bar{x})}{n(n-1)}}\right)$
1000	1.267907777E-11	3.560769267E-06	1.186923089E-06
2000	2.852852777E-14	1.689039010E-07	5.630130033E-08
3000	1.723400000E-16	1.312783302E-08	4.375944342E-09
4000	1.739569444E-18	1.318927384E-09	4.396424613E-10
5000	2.024053694E-20	1.422692410E-10	4.742308034E-11
6000	6.345816111E-23	7.966063087E-12	2.655354362E-12
7000	1.158381944E-25	3.403501056E-13	1.134500352E-13
8000	1.977760010E-27	4.447201367E-14	1.482400455E-14
9000	2.860577777E-30	1.691324267E-15	5.637747558E-16

Table IV: Variations of the convergence behavior for 100 solver runs on FV3.

# iters	averg. residual	max res	min res.	variation
1000	5.891843E-02	5.892425E-02	5.891381E-02	1.04E-05
2000	3.704219E-03	3.704469E-03	3.704032E-03	4.37E-07
3000	2.328863E-04	2.329025E-04	2.328636E-04	3.89E-08
4000	1.463948E-05	1.464075E-05	1.463681E-05	3.94E-09
5000	9.204411E-07	9.206522E-07	9.202036E-07	4.49E-10
6000	5.787439E-08	5.788572E-08	5.786295E-08	2.28E-11
7000	3.638487E-09	3.638883E-09	3.637891E-09	9.92E-13
8000	2.287199E-10	2.287934E-10	2.286609E-10	1.33E-13
9000	1.438192E-11	1.438458E-11	1.438008E-11	4.50E-15

Table V: Variations of the convergence behavior for 100 solver runs on FV3.

The experiment results, summarized in Figures 4, 5, 6, 7 and 9, show that for test systems CHEM97ZTZ, FV1, FV2, FV3 and TREFETHEN_2000 the synchronous Gauss-Seidel algorithm converges in considerably less iterations. This superior convergence behaviour is intuitively expected, since the synchronization after each component update allows the use the updated components immediately for the next update. For the Jacobi implementation, the synchronization after each iteration ensures the usage of updated components for the next iteration. Since this is not true for the asyn-

Figure 4: Convergence for test matrix CHEM97ZTZ

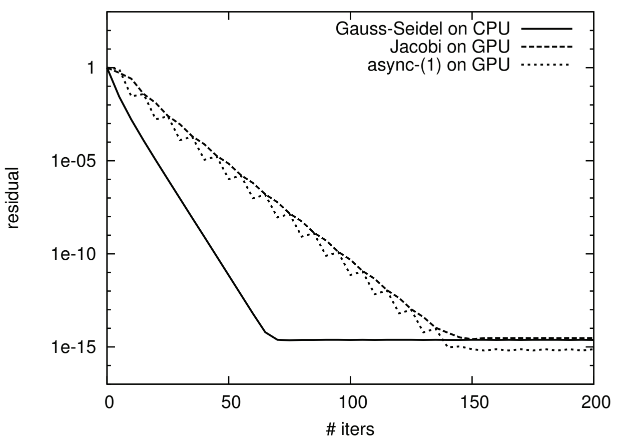


Figure 5: Convergence for test matrix FV1

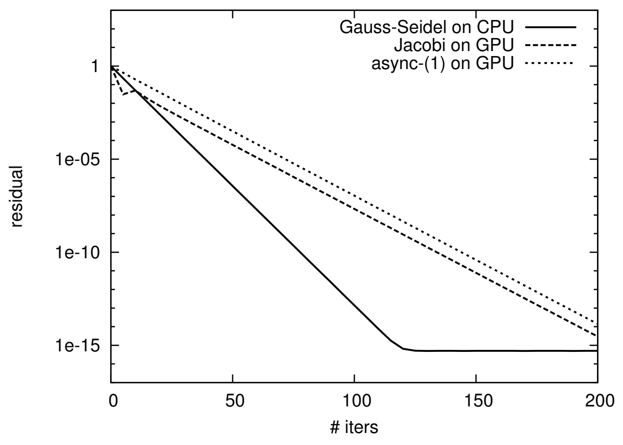


Figure 6: Convergence for test matrix FV2

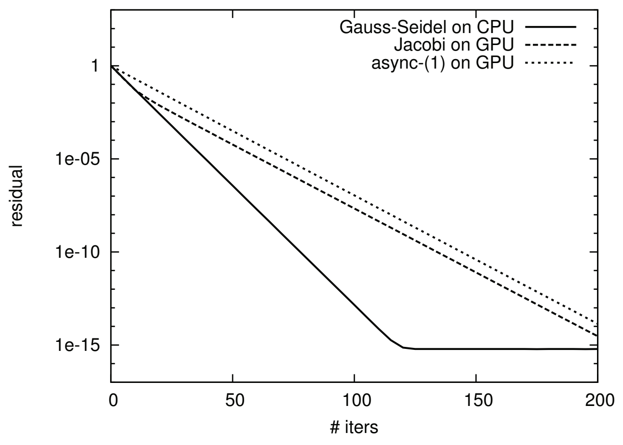


Figure 7: Convergence for test matrix FV3

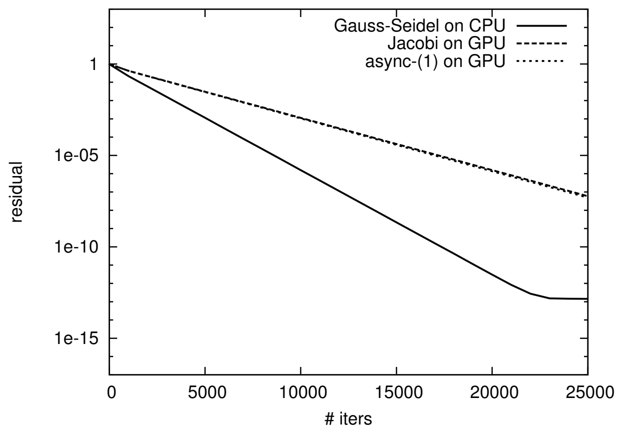


Figure 8: Convergence for test matrix S1RMT3M1

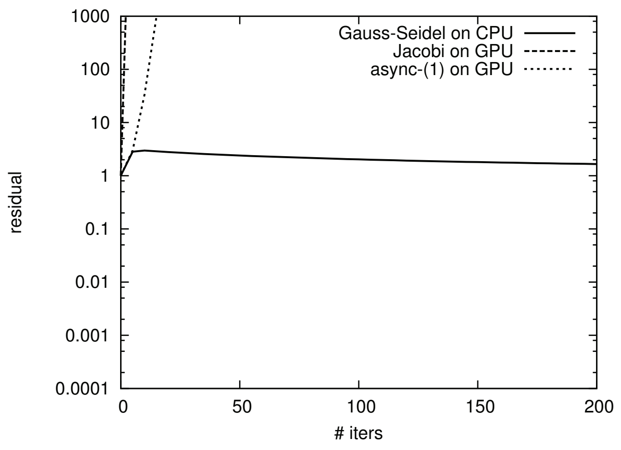
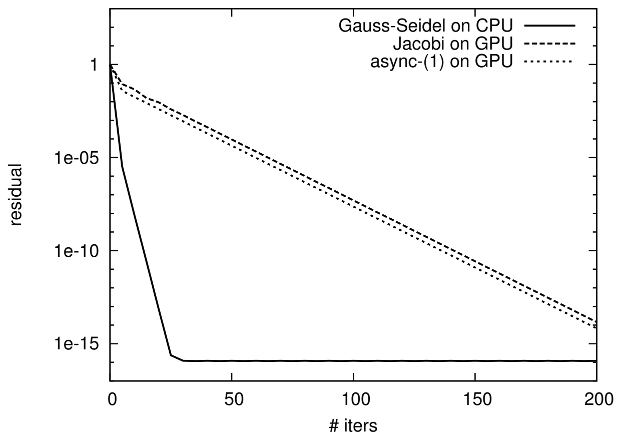


Figure 9: Convergence for test matrix TREFETHEN_2000



chronous iteration, the convergence depends on the problem and the update order – the usage of updated components implies the potential of a Gauss-Seidel convergence rate, while the chaotic properties may trigger convergence slower than Jacobi.

Still, we observe for all test cases convergence rates similar to the synchronized counterpart, which is also almost doubled compared to Gauss-Seidel.

The results for test matrix S1RMT3M1 (Figure 8) show an example where neither of the methods is suitable for direct use. The reason is that here $\rho(M) > 1$ (in particular, $\rho(M) \approx 2.65$). Nevertheless, note that this matrix is SPD and Jacobi-based methods still can be used after a proper scaling is added, e.g., taking $M = I - \tau D^{-1}A$ with $\tau = \frac{2}{\lambda_1+\lambda_n}$, where λ_1 and λ_n approximate correspondingly the smallest and largest eigenvalue of $D^{-1}A$.

C. Block-asynchronous iteration method

We now consider a block-asynchronous iteration method which additionally performs a few Jacobi-like iterations on every subdomain. A motivation for this approach is hardware

method	computation time for # global iterations 100	200	300	400	500
async-(1)	1.376425	2.437521	3.501462	4.563519	5.624792
async-(2)	1.431110	2.546361	3.660030	4.773864	5.891870
async-(3)	1.482574	2.654470	3.819478	4.987472	6.156434
async-(4)	1.532940	2.749808	3.972644	5.191812	6.410378
async-(5)	1.577105	2.838185	4.099068	5.363081	6.655686
async-(6)	1.629628	2.938897	4.255335	5.569045	6.879329
async-(7)	1.680975	3.044979	4.412199	5.778823	7.144304
async-(8)	1.736295	3.148895	4.571684	5.990520	7.409536
async-(9)	1.786658	3.259132	4.730689	6.202893	7.676786

Table VI: Overhead to total execution time by adding local iterations, matrix FV3.

related – specifically, this is the fact that the additional local iterations almost come for free (as the subdomains are relatively small and the data needed largely fits into the multiprocessors' caches). In Table VI we report the overhead triggered by the additional local iterations conducted on the subdomains. Switching from async-(1) to async-(2) affects the total computation time by less than 5%, independent of the total number of global iterations. At the same time, this leads to an algorithm where every component is updated twice as often. Even if we iterate every component locally by say 9 Jacobi iterations, the overhead is less than 35%, while the total updates for every component differ by a factor of 9. There exists though a critical point, where adding more local iterations does not improve the overall performance. It is difficult to analyze the trade-off between local and global iterations [25], and we desist from giving a general statement for the optimal choice of local iterations. This is due to the fact that the choice depends not only on the characteristics of the linear problem, but also on the iteration status of the thread block and the local components (as related to the asynchronicity), subdomain sizes, and other parameters. Based on empirical tuning and intuition (trying to match the convergence of the new method to that of a Gauss-Seidel iteration) we set the number of local Jacobi-like updates to five. Therefore we chose async-(5) for our subsequent analysis of the block-asynchronous iteration method for GPUs. From now on, the number of iterations we report is only the number of global iterations, where every single component is updated five times.

Now, we compare the convergence rate of async-(5) with the Gauss-Seidel convergence rate.

As theoretically expected, synchronous relaxation as well as the block-asynchronous async-(5) are not directly suitable to use for the S1RMT3M1 matrix. Besides this case, the async-(5) improves the convergence rate for all other test cases. In fact, as a rule of thumb, we would expect, and indeed observe as shown below, an improvement factor of about two.

The rule of thumb expectation for the convergence rate of the async-(5) algorithm is based on the rate with which values are updated and the rate of propagation for the up-

Figure 10: Convergence behavior of async-(5) for test matrix CHEM97ZTZ

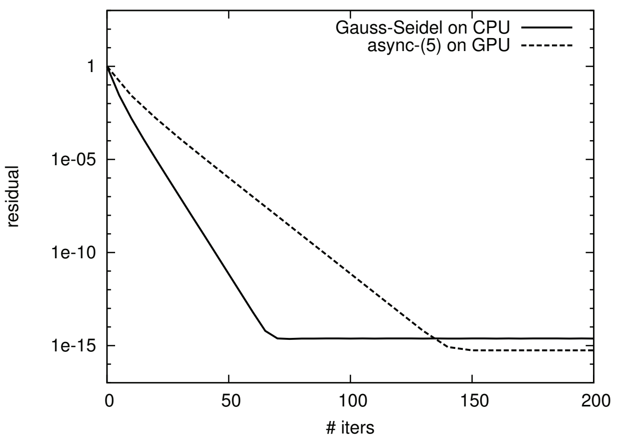


Figure 11: Convergence behavior of async-(5) for test matrix FV1

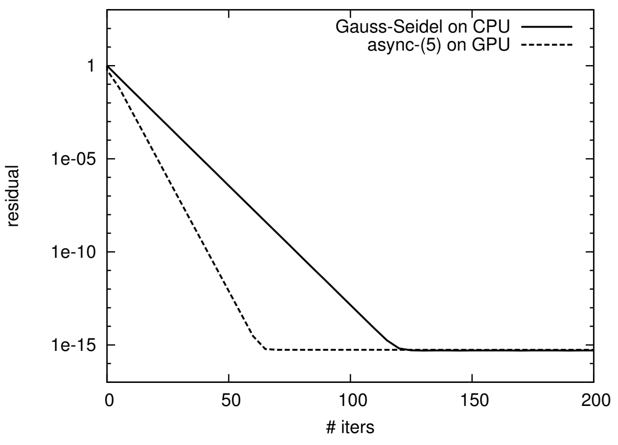


Figure 12: Convergence behaviour of async-(5) for test matrix FV2

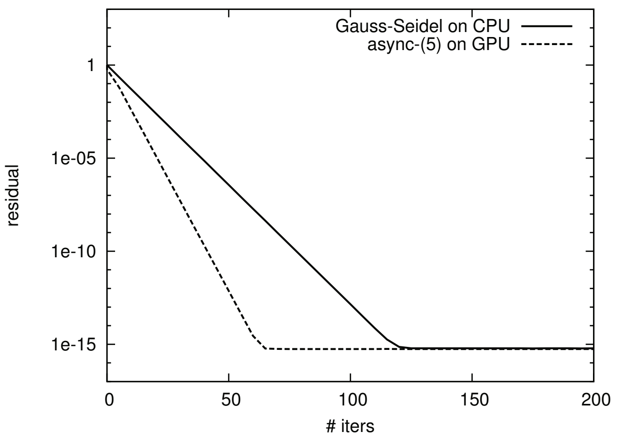


Figure 13: Convergence behavior of async-(5) for test matrix FV3

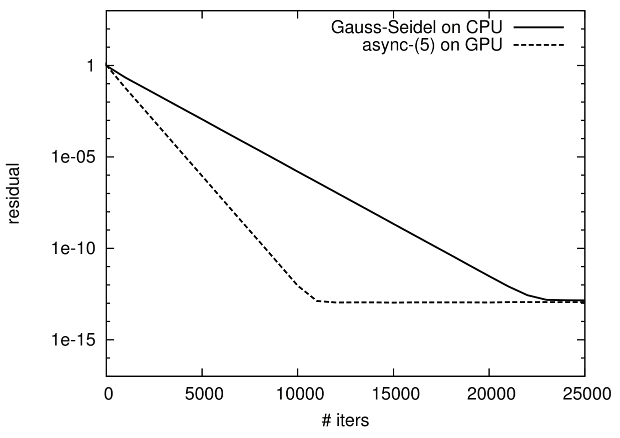


Figure 14: Convergence behavior of async(5) for test matrix S1RMT3M1

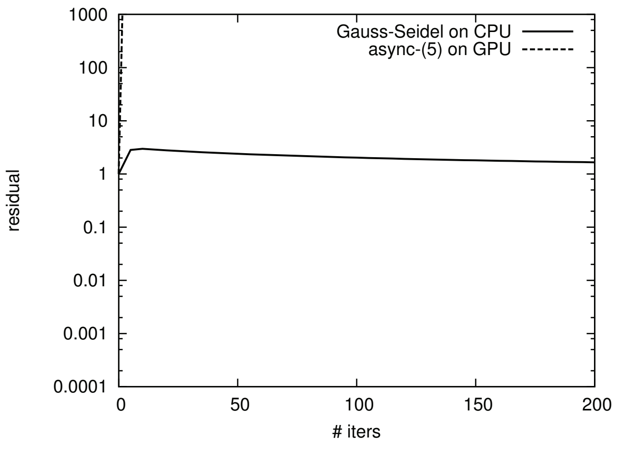
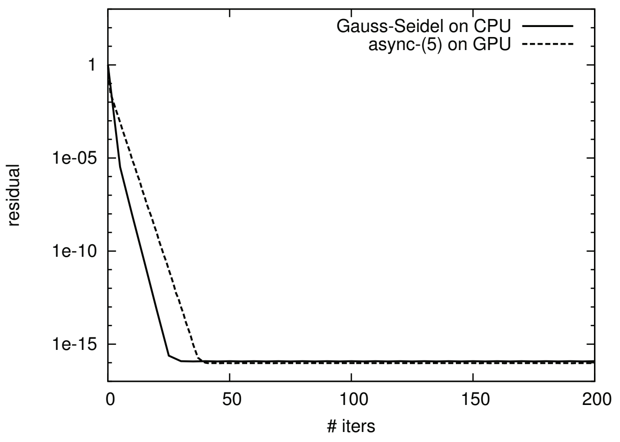


Figure 15: Convergence behavior of async-(5) for test matrix TREFETHEN_2000



dates. For example, this is the observation that Gauss-Seidel has in general twice better convergence rate than Jacobi. In other words, four Jacobi iterations would be expected (in general) to be twice better than one Gauss-Seidel iteration. The experiments show that the convergence of async-(5) for CHEM97ZTZ is characteristic for the convergence of the synchronous Jacobi iteration. This can be explained by the fact that the local matrices for CHEM97ZTZ are diagonal and therefore it does not matter how many local iterations would be performed. An improvement for this case can be obtained by reordering. The case for TREFETHEN_2000 is similar – although there is improvement compared to Jacobi, the rate of convergence for async-(5) is not twice better than Gauss-Seidel, and the reason is again the structure of the local matrices (see Figure 1 for the structure of the matrices and Figures 10 and 15 for the convergence results). Considering the remaining linear systems of equations FV1, FV2 and FV3, we obtain approximately twice faster convergence by using the async-(5) algorithm (see Figures 11, 12, and 13). Since for these cases most of the relevant matrix entries are gathered on or near the diagonal and therefore are taken into account in the local iterations on the subdomains, we observe the convergence gain expected by iterating locally. Hence, as long as the asynchronous method converges and the off-block entries are "small", adding local iterations may be used to not only compensate the convergence loss due to the chaotic behavior, but moreover to gain significant overall convergence improvements.

But the convergence rate alone does not determine, whether an iterative method is efficient or not. The second important characteristic we must analyze is the time needed to execute one iteration on the respective hardware platform. While the time can be easily measured for the synchronous iteration methods, the nature of asynchronous relaxation schemes does not allow the determination of the time needed per iteration, since not all components are updated at the same time. Hence, only an average time per global iteration can be computed by dividing the total time by the total number of iterations. Therefore, we also use an average time for the CPU implementation. It should also be mentioned, that while the average timings for one iteration on the CPU are almost constant, for the GPU implementations the iteration time differs considerably. Due to the large overhead when performing only a small number of iterations, the average computation time per iteration decreases significantly for cases where a large number of iterations is conducted. This behavior is shown in Figure 16, where the average iteration timings for the test matrix FV3 are reported. As average time per iteration for the GPU implementations, we took the average of the cases when conducting $10, 20, 30 \ldots 200$ iterations. The average timings for the Gauss-Seidel method on the CPU and the Jacobi and async-(5) iteration on the GPU are shown in Table VII. Note that the iteration time for Jacobi is slightly higher due to the synchronization after

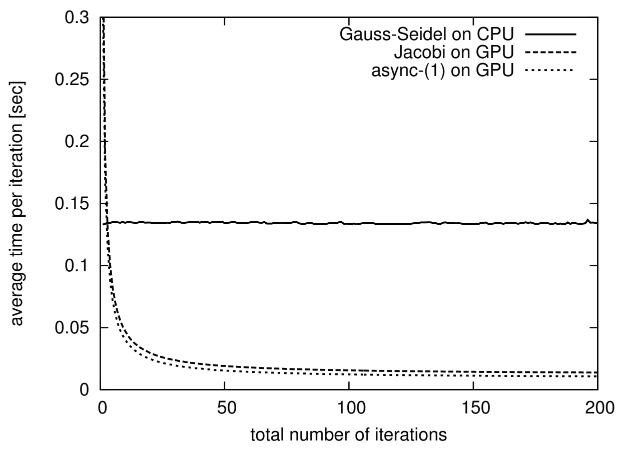


Figure 16: Average iteration timings of CPU/GPU implementations depending on total iteration number, test matrix FV3.

Matrix name	G.-S. (CPU)	Jacobi (GPU)	async-(5) (GPU)
CHEM97ZTZ	0.008448	0.002051	0.001742
FV1	0.120191	0.019449	0.012964
FV2	0.125572	0.020997	0.014729
FV3	0.125577	0.021009	0.014737
S1RMT3M1	0.039530	0.006442	0.004967
TREFETHEN_2000	0.007603	0.001494	0.001305

Table VII: Average iteration timings in seconds.

Figure 17: Time to solution for CHEM97ZTZ.

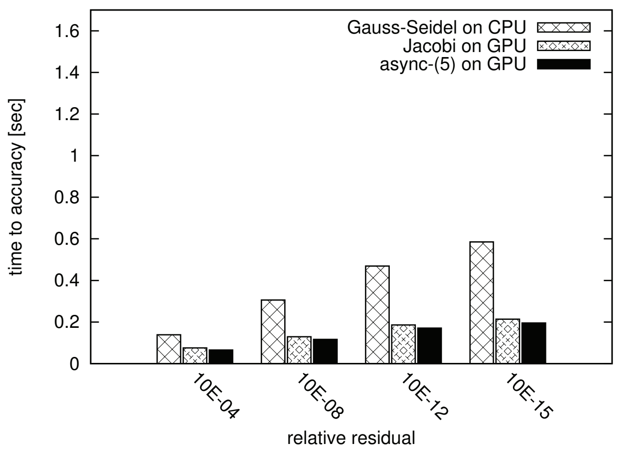
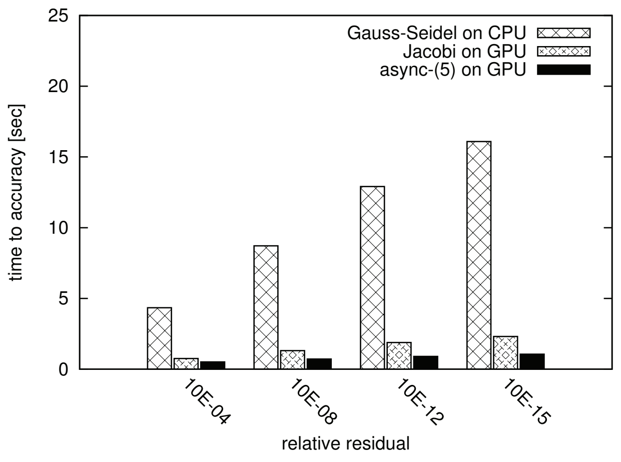


Figure 18: Time to solution for FV1.



each iteration.

Overall, we observe, that the average iteration time for the async-(5) method using the GPU is only a fraction of the time needed to conduct one iteration of the synchronous Gauss-Seidel on the CPU. While for small iteration numbers and problem sizes we have a factor of around 5, it rises over 10 for large systems and high total iteration numbers. The question is, whether the faster component updates can overcompensate the cases of slower convergence rate, i.e. the matrices CHEM97ZTZ and TREFETHEN_2000. In this case, the asynchronous method using the GPU as accelerator would still outperform the synchronous Gauss-Seidel on the CPU.

D. Performance of the block-asynchronous iteration method

To analyze the performance of the block-asynchronous iteration method, we show in Figures 17, 18 19, 20, and 21 the average time needed for the synchronous Gauss-Seidel, the synchronous Jacobi and the block-asynchronous iteration method to provide a solution approximation of certain accuracy, relative to the initial residual. We want to mention, that due to the implementation of the asynchronous method on the GPU where 10 iterations are merged into a stream of kernels, over-relaxation is possible. This means, that the demanded accuracy could have been achieved earlier by conducting a smaller number of iterations, but since the overhead of at most 9 additional iterations is small, this issue has no significant impact on the performance of the method. Since the convergence of the Gauss-Seidel method for S1RMT3M1 is almost negligible, and the Jacobi and the asynchronous iteration does not converge at all, we limit this analysis to the linear systems of equations CHEM97ZTZ, FV1, FV2, FV3 and TREFETHEN_2000.

We observe, that for CHEM97ZTZ, FV1, FV2, and FV3 async-(5) using the GPU is able to provide the solution approximation of the demanded accuracy in a considerably shorter time frame than the synchronous Gauss-Seidel or Jacobi. Especially for high accuracy approximations, the speedup gained by the block-asynchronous GPU implementation is significant. This is due to the fact that for these high iteration numbers the overhead triggered by memory transfer and GPU kernel call has minor impact. Hence, the cases of slower convergence rate can be overcompensated by the higher computational power that is used. For linear equation systems with considerable off-diagonal part e.g. CHEM97ZTZ, the improvement compared to Jacobi becomes smaller since the entries outside the subdomains are

Figure 19: Time to solution for FV2.

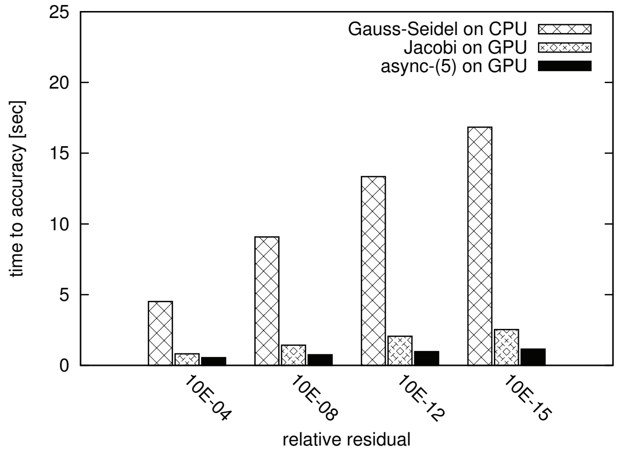


Figure 20: Time to solution for FV3.

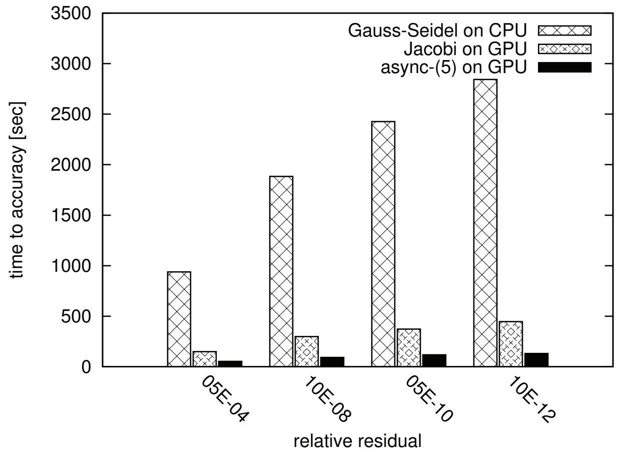
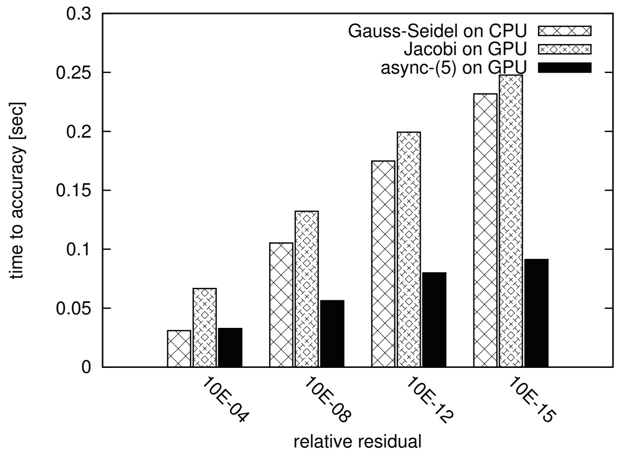


Figure 21: Time to solution for TREFETHEN_2000.



not taken into account for the local iterations. Still, due to the faster kernel execution the block-asynchronous iteration provides the solution approximation in shorter time.

For TREFETHEN_2000, the async-(5) method using the GPU does not reach the Gauss-Seidel performance on the CPU for small iteration numbers. This is due to the characteristics of the test matrix: the linear system combines small dimension with low condition number, enabling fast convergence. In this case the overhead triggered by the GPU kernel calls is crucial. But as the difference between the execution times is small, applying the GPU-accelerated asynchronous scheme would not be fatal. Going to higher iteration numbers, also for this test case, the async-(5) outperforms the CPU implementation of Gauss-Seidel.

As already stated, using Jacobi-type iterations is not a good choice for the S1RMT3M1 problem. Although the sufficient for convergence condition is not satisfied, the synchronous Gauss-Seidel converges slowly but neither the Jacobi nor the asynchronous method converges. Hence, using asynchronous methods is only reasonable as long as the convergence can be guaranteed. Otherwise, it is a statistical question whether the asynchronous scheme will provide a solution approximation in a shorter time, since it may always be possible, that the ”right” order of component updates is chosen, leading to a synchronous version of the algorithm.

We conclude from this analysis, that asynchronous iteration schemes using parallel devices have to be used carefully, but for suitable systems they may trigger considerable performance increase when solving linear systems of equations.

V. CONCLUSIONS

We developed asynchronous relaxation methods for highly parallel architectures. The experiments have revealed the potential of using them on GPUs. The absence of synchronization points enables not only to reach a high scalability, but also to efficiently use the GPU architecture. Adding local refinement steps into the GPU kernels improves the convergence without impacting the execution time of a global iteration. As a result, for all sufficiently large test cases where the asynchronous iteration converged, the block-asynchronous iteration method (with 5 subdomain iterations) improves two times the convergence rate of the effective but difficult to parallelize Gauss-Seidel method, while retaining the high-performance and scalability enabled by its asynchronous nature. Nevertheless, the numerical properties of asynchronous iteration pose some restrictions on the usage. The conditions for their convergence limit the area where asynchronous iteration methods can be applied. Since the matrix characteristics are in general not known a priori, e.g., in purely algebraic methods, it seems reasonable to apply asynchronous iteration only to cases where the matrix properties are known/expected (or can be derived easily on the "fly"), e.g., matrices derived from particular PDE discretizations. Therefore, one focus of future research will be on using asynchronous iterations schemes for cases where their synchronous counterparts are known to work.

The presented approach could be embedded in a multigrid framework, replacing the traditional Gauss-Seidel based smoothers. However it is still a subject of further research how to determine the optimal number for the various parameters arising in the asynchronous methods, such as number of local iterations, subdomain sizes, scaling parameters, etc., with respect to the problem. This optimization may not only be dependent on the problem, but also on the parameters in the multigrid framework like prolongation operator and number of pre- and post-smoothing steps, and the used hardware system.

Acknowledgment

Implementation and computations were done at the Innovative Computing Laboratory at the University of Tennessee. The authors would like to thank the National Science Foundation, the Department of Energy, NVIDIA, and Microsoft Research for supporting this research effort.

References

[1] J. Dongarra *et al*, "The international ExaScale software project roadmap," *Int. J. of High Performance Computing & Applications*, vol. 25, no. 1, 2011.

[2] *TESLA C2050 / C2070GPU Computing Processor*, NVIDIA Corporation, July 2010.

[3] *AMD Radeon HD 6990 Graphics*, Advanced Micro Devices (AMD), 2011, product Specifications.

[4] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.

[5] Y. Saad and M. H. Schultz, "GMRES: A generalized minimal residual method for solving nonsymmetric linear systems." 1986.

[6] A. T. Chronopoulos and A. B. Kucherov, "A parallel krylov-type method for nonsymmetric linear systems."

[7] C. T. Kelley, *Iterative Methods for Linear and Nonlinear Equations*. SIAM, 1995.

[8] J. A. H. Courtecuisse, "Parallel dense gauss-seidel algorithm on many-core processors," Jun. 2009.

[9] R. Bagnara, "A unified proof for the convergence of jacobi and gauss-seidel methods," *SIAM Rev.*, vol. 37, pp. 93–97, March 1995.

[10] D. Chazan and W. Miranker, "Chaotic Relaxation," *Linear Algebra and Its Applications*, vol. 2, no. 7, pp. 199–222, 1969.

[11] A. Frommer and D. B. Szyld, "On asynchronous iterations," *Journal of Computational and Applied Mathematics*, vol. 123, pp. 201–216, 2000.

[12] A. Frommer, H. Schwandt, and D. B. Szyld, "Asynchronous weighted additive Schwarz methods," *Electronic Transactions on Numerical Analysis*, vol. 5, pp. 48–61, 1997.

[13] D. B. Szyld, "The mystery of asynchronous iterations convergence when the spectral radius is one," Department of Mathematics, Temple University, Philadelphia, Pa., Tech. Rep. 98-102, October 1998, available at `http://www.math.temple.edu/szyld`.

[14] U. Aydin and M. Dubois, "Generalized asynchronous iterations," pp. 272–278, 1986.

[15] ——, "Sufficient conditions for the convergence of asynchronous iterations," *Parallel Computing*, vol. 10, no. 1, pp. 83–92, 1989.

[16] D. P. Bertsekas and J. Eckstein, "Distributed asynchronous relaxation methods for linear network flow problems," *Proceedings of IFAC '87*, 1986.

[17] J. C. Strikwerda, "A convergence theorem for chaotic asynchronous relaxation," *Linear Algebra and its Applications*, vol. 253, no. 1-3, pp. 15–24, Mar. 1997.

[18] G. M. Baudet, "Asynchronous iterative methods for multiprocessors," *J. ACM*, vol. 25, no. 2, pp. 226–244, 1978.

[19] "Intel C++ Compiler Options," Intel Corporation, document Number: 307776-002US.

[20] *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 2nd ed., NVIDIA Corporation, August 2009.

[21] *CUDA TOOLKIT 4.0 READINESS FOR CUDA APPLICATIONS*, 4th ed., NVIDIA Corporation, March 2011.

[22] Z.-Z. Bai, V. Migallón, J. Penadés, and D. B. Szyld, "Block and asynchronous two-stage methods for mildly nonlinear systems," *Numerische Mathematik*, vol. 82, pp. 1–20, 1999.

[23] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. Meier Yang, "Multigrid smoothers for ultra-parallel computing," 2011, lLNL-JRNL-435315.

[24] 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*.

[25] U. Meier Yang, "On the use of relaxation parameters in hybrid smoothers," *Numerical Linear Algebra with Applications*, vol. 11, pp. 155–172, 2011.