A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem on a Multi-GPU Platform

M. Ament[†], G. Knittel[‡], D. Weiskopf[†], W. Straßer[‡]

[†] VISUS Visualization Research Center
Universität Stuttgart
Stuttgart, Germany
[marco.ament|weiskopf]@visus.uni-stuttgart.de

[‡] WSI/GRIS
Universität Tübingen
Tübingen, Germany
[knittel|strasser]@gris.uni-tuebingen.de

Abstract - **We present a parallel conjugate gradient solver for the Poisson problem optimized for multi-GPU platforms. Our approach includes a novel heuristic Poisson preconditioner well suited for massively-parallel SIMD processing. Furthermore, we address the problem of limited transfer rates over typical data channels such as the PCI-express bus relative to the bandwidth requirements of powerful GPUs. Specifically, naive communication schemes can severely reduce the achievable speedup in such communication-intense algorithms. For this reason, we employ overlapping memory transfers to establish a high level of concurrency and to improve scalability. We have implemented our model on a high-performance workstation with multiple hardware accelerators. We discuss the mathematical principles, give implementation details, and present the performance and the scalability of the system.**

I. Introduction

In parallel and distributed systems, graphics processors have become a serious competitor for classical CPU environments. The high performance concerning floating point operations offers attractive capabilities for general purpose computations on GPUs (GPGPU). Therefore, numerical algorithms can be substantially accelerated as long as the algorithms map well to the specific characteristics of the hardware. On the other hand, there is no comparable data transfer technology that keeps up with this development, hence the already existing gap between bandwidth and calculation speed is widening further.

In communication- and bandwidth-limited problems, like the solution of a sparse linear system of equations, this causes poor or even negative speedups, at least as long as the problem size remains small. This in turn leads to low scalability and thereby renders a multi-GPU solution questionable. Our work focuses on the Poisson problem that arises in a wide range of applications and disciplines like computational fluid dynamics, gravitational physics, electrostatics, magnetostatics, or image editing. Numerical solvers typically lead to discretizations with a huge amount of data that must be processed by the system, especially in 3D. Therefore, iterative solvers are usually employed instead of direct methods.

A commonly used and well examined iterative algorithm for solving sparse symmetric positive definite linear systems is the conjugate gradient (CG) method. In conjunction with an appropriate preconditioner (PCG), the method has proven to be an excellent approach in a wide spectrum of applications.

Although highly sophisticated multigrid solvers are usually faster, PCG methods remain an important alternative because they are easy to implement and can handle arbitrarily complex boundaries naturally without complicated extensions [18].

However, a direct and efficient mapping to parallel computation on architectures with multiple GPUs is difficult due to the specific performance characteristics of GPUs. In particular, memory access schemes, latency, and bandwidth exhibit behaviors different from traditional CPU-based systems. Our goal is to develop a preconditioned CG solver that is suited for multi-GPU architectures. In particular, the issue of efficient preconditioning has been largely ignored in the previous GPU literature. Therefore, we have developed new strategies to achieve reasonable speedups with preconditioning and algorithmic latency hiding.

Widely used preconditioners like incomplete Cholesky (IC) factorization or symmetric successive over-relaxation (SSOR) are hard to parallelize on graphics hardware because of the inherently serial processing of the triangular systems in the forward and backward substitution. On the other hand, simple preconditioners like Jacobi are easy to parallelize but have only minor impact on the speed of convergence. We seek a new method for the Poisson problem that is as easy to implement as Jacobi but shows significant improvements in the computation time, especially on current GPUs.

II. Related Work

The parallel solution of linear systems of equations is a well examined but still active field in high-performance computing. A good introduction to parallel numerical algebra and state-of-the-art iterative methods for sparse systems can be found in Saad's textbook [21], which covers basic operations like sparse matrix-vector multiplication (SpMV) and also provides an introduction to Krylov

subspace methods with preconditioners like Jacobi, IC, or (S)SOR.

A detailed study on and optimizations for the cost-intense SpMV operation can be found in the paper by Lee et al. [15]. The authors present an approach for the selection of tuning parameters, based on empirical modeling and search, that consists of an off-line benchmark, a runtime search and a heuristic performance model.

The CG method was introduced by Hestenes et al. [12] and has a long and successful history in computational mathematics (e.g. [9]). Another source for the CG method is Shewchuk [22], who provides additional background on the derivation.

The rate of convergence was examined by Van der Sluis and Van der Horst [23], who also showed that the CG method has superlinear behavior when one or more extreme Ritz values are close enough to the corresponding eigenvalues.

The distribution of data on multiple nodes causes large communication overhead in the SpMV operations. Sparse matrices usually arise from finite discretizations of partial differential equations (PDEs) and describe the adjacency of a grid or mesh; hence a domain decomposition usually yields independent areas that only require local data access. The report by Demmel et al. [6] shows different variants of the PCG algorithm in order to facilitate data transfers overlapping with computation. In this way, the expensive communication is partially hidden behind the processing of the independent areas.

Using graphics hardware for non-graphics applications has become a common approach to accelerate compute-intense tasks [8]. A good overview of GPGPU computing can be found in [16] and [10]. An early implementation of common matrix-matrix multiplications was presented by Larsen and McAllister [14]. In order to improve the performance of the previous work, Hall et al. [11] proposed a multi-channel algorithm to increase cache coherence and reduce memory bandwidth.

A more general approach is followed by Krüger and Westermann [13], who implemented a framework of numerical simulation techniques on graphics hardware. The authors presented data structures for sparse matrices in texture memory as well as basic linear algebra operations like a matrix-vector product or a vector reduction. They also implemented an unpreconditioned CG solver on the GPU in order to show the efficiency of their approach.

Direct solvers like Gauss-Jordan elimination and LU decomposition were implemented on the GPU by Galoppo et al. [7] for dense linear systems. The performance was comparable to optimized CPU code.

For huge sparse linear systems, iterative methods are usually more preferable. Bolz et al. [2] presented both a CG and a multigrid implementation on graphics hardware to solve PDEs on regular and unstructured grids. The authors were able to achieve a speedup of about 2 compared to CPU implementations for problem sizes of 513×129 in the matrix multiply computation. Another GPU implementation of CG was presented by Wiggers et al. [24]. In performance measurements, an NVIDIA G80 GPU outperformed a dual-core Woodcrest CPU by a factor of 2.56. However, preconditioning was not addressed in either work.

In a more recent publication, Buatois et al. [4] introduced their framework CNC for solving general sparse linear systems on the GPU with a Jacobi-preconditioned CG method. According to their results, a speedup of 6 was achieved compared to an SSE3-CPU implementation.

Current developments in graphics hardware seem to follow the trend of multiple GPU cores in a single machine. So far, implementations of CG on these platforms are very rare. Cevahir et al. [20] published a mixed precision iterative refinement algorithm for general matrices. They achieved up to 11.6 GFlops in the matrix-vector multiplication on a single GeForce 8800 GTS card and their CG implementation reached up to 24.6 GFlops with four GPUs but preconditioning was also not addressed by the authors.

III. Basics

A. The Poisson Equation and its Applications

The Poisson equation is a second-order PDE that arises in a wide range of physical problems. Its general form is

$$\Delta\phi = f \tag{1}$$

where Δ denotes the Laplacian while ϕ and f are complex- or real-valued functions on a manifold. In Euclidean space and with three-dimensional Cartesian coordinates, the Poisson equation can be formulated as

$$\Delta\phi = \nabla^2\phi = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}\right) \cdot \phi = f \tag{2}$$

Applications where this equation occurs include computational fluid dynamics, and the computation of potentials such as in Maxwell's equations or in gravitational physics. Our work is motivated by, but not limited to, incompressible fluid simulations and the Navier-Stokes equations with the following Poisson equation (3), which must be solved to project the velocity field $\boldsymbol{u}^*$ on its solenoidal part [3]. In this specific example, p denotes the pressure, ρ the density, and Δt the time step.

$$\Delta p = \frac{\rho}{\Delta t}(\nabla \cdot \boldsymbol{u}^*) := \mathrm{b} \tag{3}$$

B. Discretization

Analytical solutions for the Poisson equation are usually not feasible. A common approach is to discretize the

domain into a finite number of grid points and to approximate the derivatives in these points with finite differences. For this reason, we divide our domain of interest into an equidistant regular grid in three dimensions, which is a widely used procedure. The Laplacian is approximated with backward differences, which leads to a discrete formulation for every grid cell $p_{x,y,z}$. In the same way, the right-hand side of the Poisson equation is discretized as well, denoted with $b_{x,y,z}$. Hence, the discrete Poisson equation for an inner cell is

$$\Delta p \approx \frac{6p - p_{x+1} - p_{x-1} - p_{y+1} - p_{y-1} - p_{z+1} - p_{z-1}}{\Delta x^2} = b \quad (4)$$

For simplicity, we omitted unmodified indices that correspond to x,y,z. If the domain is subdivided into $n = s_x \cdot s_y \cdot s_z$ cells, an $n \times n$ sparse linear system must be solved. In the following, we need to define a compact format to describe the coefficients c of each row in the matrix. For this reason, we use the following representation for the i-th row

$$\text{row}_i = (c_{z-1}, c_{y-1}, c_{x-1}, c, c_{x+1}, c_{y+1}, c_{z+1}) \quad (5)$$

C. The PCG Algorithm in a Nutshell

The preconditioned conjugate gradient method is a well known iterative algorithm for solving linear systems [22]. The matrix is required to be symmetric positive definite for the CG method to converge. The above described matrix is well studied and satisfies all constraints. With given input vectors $\mathbf{p}_0$ and $\mathbf{b}$, the solution of A$\mathbf{p}$ = $\mathbf{b}$ with the PCG algorithm and a preconditioner M is described as

$$\begin{array}{ccc} \mathbf{r}_0 = \mathbf{b} - A\mathbf{p}_0 & \mathbf{h}_0 = M^{-1}\mathbf{r}_0 & \mathbf{d}_0 = \mathbf{h}_0 \\ & r_{\text{old}} = \langle \mathbf{r}_0, \mathbf{h}_0 \rangle \quad i = 1 & \end{array} \quad (6)$$

While $i < i_{max}$ and $r_{old} > \varepsilon$ do

$$\begin{array}{ll} \mathbf{t} = A\mathbf{d}_{i-1} & \alpha = \dfrac{r_{\text{old}}}{\langle \mathbf{d}_{i-1}, \mathbf{t} \rangle} \\ \mathbf{r}_i = \mathbf{r}_{i-1} - \alpha\mathbf{t} & \mathbf{p}_i = \mathbf{p}_{i-1} + \alpha\mathbf{d}_{i-1} \\ \mathbf{h}_i = M^{-1}\mathbf{r}_i & r_{\text{new}} = \langle \mathbf{r}_i, \mathbf{h}_i \rangle \\ \beta = \dfrac{r_{\text{new}}}{r_{\text{old}}} & \mathbf{d}_i = \mathbf{h}_i + \beta\mathbf{d}_{i-1} \\ r_{\text{old}} = r_{\text{new}} & i = i + 1 \end{array} \quad (7)$$

D. State-of-the-Art Preconditioning on the GPU

The concept of preconditioning can dramatically accelerate the CG method because one of its properties is that the convergence rate does not only depend on n but also on the condition number of the system. A matrix is called well conditioned if it is close to the identity matrix. For symmetric positive definite matrices, the condition number κ is defined as

$$\kappa(A) = \frac{\lambda_{\max}}{\lambda_{\min}} \quad (8)$$

with the maximum and minimum eigenvalues $\lambda_{\max}$ and $\lambda_{\min}$. Note that the identity matrix has a value of $\kappa = 1$. The closer an arbitrary matrix is to the identity, the faster the CG method will converge. We will use this property later to show that our new approach for a Poisson preconditioner is reasonable.

The objective of preconditioning is to transform the original system into an equivalent system with the same solution, but a lower condition number. However, the computational overhead of applying the preconditioner must not cancel out the benefit of fewer iterations. Common preconditioners like IC or SSOR are well suited for CPU processing and can heavily reduce the computation time. So far, these high-performance preconditioners have not been ported successfully to GPU architectures. It is crucial to understand which part of the algorithms are hard to parallelize on GPUs and this was also the basis and motivation for developing our new approach.

In the following, we assume an SSOR preconditioner M. The computation of M can be performed efficiently on the GPU. By decomposing the matrix A into its lower triangular part L and its diagonal D

$$A = L + D + L^T \quad (9)$$

the SSOR preconditioner is given by

$$M(\omega) = \frac{1}{2-\omega}\left(\frac{1}{\omega}D + L\right)\left(\frac{1}{\omega}D\right)^{-1}\left(\frac{1}{\omega}D + L\right)^T \quad (10)$$

Applying the preconditioner to the PCG method, on the other hand, is an inherently serial procedure and does not map well to GPU platforms. This arises from the approach to approximate the original system A with M. Unfortunately the PCG algorithm as described above requires the inverse M^{-1}. In general, these preconditioners are symmetric positive definite and are designed in such a way that a decomposition of M can be easily obtained with

$$M = KK^T \quad (11)$$

For example, in the case of SSOR

$$K = \frac{1}{\sqrt{2-\omega}}\left(\frac{1}{\omega}D + L\right)\left(\frac{1}{\omega}D\right)^{-\frac{1}{2}} \quad (12)$$

Instead of calculating the inverse of M, the preconditioner is applied by solving

$$M\mathbf{h}_i = KK^T\mathbf{h}_i = \mathbf{r}_i \quad (13)$$

for $\mathbf{h}_i$. This is achieved by solving two triangular systems with forward and backward substitution

$$\text{solve} \qquad K\mathbf{q} = \mathbf{r}_i \qquad \text{and} \qquad K^T\mathbf{h}_i = \mathbf{q}\,. \tag{14}$$

On GPU architectures, these calculations are extremely slow because of the global dependency in the triangular structures. In fact, a single GPU thread is supposed to handle both triangular systems in order to gain the full benefit of decreased iteration counts. But this means that almost the entire GPU runs on idle and a great amount of the computation power is wasted. Another disadvantage is data processing because memory access is solely performed by this single thread and latency hiding is not applicable. We have implemented this strategy and present results on performance in section VI.

A widely used strategy to parallelize these solution steps on the CPU is to subdivide the matrices into independent blocks and solve the uncoupled triangular systems. This modification of the matrix yields a loss of information, which in turn leads to a slightly increased iteration count. While this is a reasonable approach on multi-CPU platforms with a comparably small number of CPUs, an uncoupling in the scale of massively-parallel GPU processing degenerates the system. Although the algorithm might still converge, the preconditioning gets counterproductive. We have also implemented this strategy and show performance figures in section VI.

The disadvantages of the triangular solution steps motivated our approach. In order to gain a real benefit from preconditioning on the GPU, we decided to develop a method that does not involve such steps. The only way to circumvent these expensive operations is to find an inverse $M^{-1} \approx A^{-1}$ directly without going the detour through an approximation of A. In general, this is called Sparse Approximate Inverse (SpAI) and was introduced by Cosgrove et al. [5]. Another class of such preconditioners is the AINV algorithm, which is based on biconjugation [1].

IV. Incomplete Poisson Preconditioner

A. Introduction

While the SpAI and AINV algorithms are reasonable approaches for arbitrary matrices, we seek an easier way for the Poisson equation. For this reason, we developed a heuristic preconditioner that approximates the inverse of the aforementioned matrix. Our goal was to find an algorithm that is as easy to implement as a Jacobi preconditioner and that is also well suited for GPU processing. This section is organized as follows. Firstly, we provide an analytical expression for the preconditioner and show that it satisfies the requirements for convergence in the PCG method. Secondly, we enforce the sparsity pattern of A on our preconditioner and sketch an informal proof that the condition of the modified system is improved compared to the original matrix. Furthermore, we provide a closed formula for the computation of the coefficients of the preconditioner that offers the possibility of a matrix-free implementation for large-scale problems.

Just like SSOR, our preconditioner only depends on the sum decomposition of A into its lower triangular part L and its diagonal D. The approximation of the inverse then is

$$M^{-1} = (I - LD^{-1})(I - D^{-1}L^T) \tag{15}$$

In this notation, I is the identity and D^{-1} can be obtained by a reciprocal operation in each diagonal element of D. Applying a preconditioner to the PCG algorithm requires that the modified system is still symmetric positive definite, which in turn requires that the preconditioner is a symmetric real-valued matrix.

$$\begin{aligned}(M^{-1})^T &= ((I - LD^{-1})(I - D^{-1}L^T))^T \\ &= (I - D^{-1}L^T)^T(I - LD^{-1})^T \\ &= (I^T - (D^{-1}L^T)^T)(I^T - (LD^{-1})^T) \\ &= (I - L^{T^T}D^{-1^T})(I - D^{-1^T}L^T) \\ &= (I - LD^{-1})(I - D^{-1}L^T) \\ &= M^{-1}\end{aligned} \tag{16}$$

Therefore, we can write our preconditioner as

$$M^{-1} = KK^T \tag{17}$$

with

$$K = I - LD^{-1} \qquad \text{and} \qquad K^T = I - D^{-1}L^T \tag{18}$$

This shows that the PCG algorithm converges when applying this preconditioner.

In order to demonstrate that the introduced method is advantageous, we provide a short abstract why the condition of the modified system improves. To make things clearer, we revert to a two-dimensional regular discretization and we only focus on an inner grid cell. In this case, the stencil of the i-th row of A is

$$\begin{aligned}\text{row}_i(A) &= (a_{y-1}, a_{x-1}, a, a_{x+1}, a_{y+1}) \\ &= (-1, -1, 4, -1, -1)\end{aligned} \tag{19}$$

Hence, we get the stencils for L, D^{-1}, and L^T

$$\begin{aligned}\text{row}_i(L) &= (-1, -1, 0, 0, 0) \\ \text{row}_i(D^{-1}) &= \left(0, 0, \tfrac{1}{4}, 0, 0\right) \\ \text{row}_i(L^T) &= (0, 0, 0, -1, -1)\end{aligned} \tag{20}$$

In the next step, we perform the operations for

$$K = I - LD^{-1} \qquad \text{and} \qquad K^T = I - D^{-1}L^T\,. \tag{21}$$

We get

$$\text{row}_i(K) = \left(\frac{1}{4}, \frac{1}{4}, 1, 0, 0\right)$$
$$\text{row}_i(K^T) = \left(0, 0, 1, \frac{1}{4}, \frac{1}{4}\right) \tag{22}$$

The final step is the matrix-matrix product KK^T, which is the multiplication of a lower and an upper triangular matrix. Each of the 3 coefficients in $\text{row}_i(K)$ hits 3 coefficients in K^T but in different columns. The interleaved arrangement in such a row-column product introduces new non-zero coefficients in the result. The stencil of the inverse increases to

$$\begin{aligned}\text{row}_i(M^{-1}) &= (m_{y-1}, m_{x+1,y-1}, m_{x-1}, m, \\ &\quad m_{x+1}, m_{x-1,y+1}, m_{y+1}) \\ &= \left(\frac{1}{4}, \frac{1}{16}, \frac{1}{4}, \frac{9}{8}, \frac{1}{4}, \frac{1}{16}, \frac{1}{4}\right)\end{aligned} \tag{23}$$

Without going into too much detail here, the stencil enlarges to up to 13 non-zero elements in three dimensions for each row, which would almost double the computational effort in a matrix-vector product compared to the 7-point stencil in the original matrix. By looking again at the coefficients in $\text{row}_i(M^{-1})$ it can be observed that the additional non-zero values $m_{x+1,y-1}$ and $m_{x-1,y+1}$ are rather small compared to the rest of the coefficients. Furthermore, this nice property remains true in three dimensions, which encouraged us to use an incomplete stencil because we assumed that these small coefficients only have a minor influence on the condition. For this reason, we set them to zero and obtain the following 5-point stencil in two dimensions

$$\text{row}_i(M^{-1}) = \left(\frac{1}{4}, 0, \frac{1}{4}, \frac{9}{8}, \frac{1}{4}, 0, \frac{1}{4}\right) \tag{24}$$

Another important property of the incomplete formulation is the fact that symmetry is still preserved as the cancellation always affects two pair-wise symmetric coefficients namely

$$(m_{x+1,y-1}, m_{x-1,y+1})$$

in two dimensions and

$$(m_{x+1,z-1}, m_{x-1,z+1})$$
$$(m_{x+1,y-1}, m_{x-1,y+1})$$
$$(m_{y+1,z-1}, m_{y-1,z+1})$$

in three dimensions. From this it follows that the CG method still converges and we call this novel preconditioner the **Incomplete Poisson (IP)** preconditioner.

B. Heuristic Analysis

So far, we have not studied why this simple scheme improves the condition of the system. Unfortunately, there is no feasible way to obtain an analytic estimate for the eigenvalues of the modified system, especially not for arbitrary boundary conditions. Therefore, we provide a heuristic approach to demonstrate the usefulness of our preconditioner and corroborate our assumption with numerical results in section VI. A perfectly conditioned matrix is the identity matrix, hence we simply try to evaluate how close our modified system reaches this ideal. For this purpose, we calculate AM^{-1}. For an inner grid cell, this leads to the following not trivially vanishing elements

$$\begin{aligned}\text{row}_i(AM^{-1}) = \Big(&-\frac{1}{4}, 0, 0, -\frac{1}{2}, -\frac{1}{8}, -\frac{1}{2}, -\frac{1}{4}, -\frac{1}{8}, \frac{7}{2}, \\ &-\frac{1}{8}, -\frac{1}{4}, -\frac{1}{2}, -\frac{1}{8}, -\frac{1}{2}, 0, 0, -\frac{1}{4}\Big)\end{aligned} \tag{25}$$

Note that this row represents the whole band in the matrix with $7/2$ as diagonal element. All elements to the left and right of this tuple are zero. This result still does not look like the identity but there is another property of the condition number that allows us to multiply the system with an arbitrary scalar value, except zero, because multiplying a matrix with a scalar does not affect the ratio of the maximum and minimum eigenvalues. For example, $\alpha \cdot AM^{-1}$ also scales all of the eigenvalues of AM^{-1} with α

$$\kappa(\alpha \cdot AM^{-1}) = \frac{\alpha \cdot \lambda_{max}}{\alpha \cdot \lambda_{min}} = \frac{\lambda_{max}}{\lambda_{min}} = \kappa(AM^{-1}) \tag{26}$$

Hence, we can multiply the above result with the reciprocal diagonal element and we get

$$\begin{aligned}\frac{2}{7} \cdot \text{row}_i(AM^{-1}) = \Big(&-\frac{1}{14}, 0, 0, -\frac{1}{7}, -\frac{1}{28}, -\frac{1}{7}, -\frac{1}{14}, -\frac{1}{28}, 1, \\ &-\frac{1}{28}, -\frac{1}{14}, -\frac{1}{7}, -\frac{1}{28}, -\frac{1}{7}, 0, 0, -\frac{1}{14}\Big)\end{aligned} \tag{27}$$

The result shows that the element-wise signed distance to the identity is much smaller than with the original Poisson system, suggesting a lower condition number. Note that the factor of $2/7$ was used for demonstration only and is not considered in the later algorithm.

Until now, we have not considered boundary conditions because they are difficult to treat. We have carried out a few experiments with small arrangements that have a high ratio of boundary to inner cells. For example, we have set up a Poisson system arising from a two-dimensional fluid simulation on a grid of 4×4 cells surrounded by 15 solid wall cells and 1 free surface cell (to make the system invertible) which means the fluid is enclosed by a

solid box but can flow in or out of the box through the free surface cell (see [3] for more details). We have calculated the maximum and minimum eigenvalues of the 16×16 matrix AM^{-1} for the unpreconditioned CG (which simply is A), for the Jacobi and SSOR preconditioner as well as for our approach.

TABLE I: CONDITION NUMBERS CALCULATED WITH MATHEMATICA [17]

	λ_{max}	λ_{min}	$\kappa = \lambda_{max} / \lambda_{min}$
CG	6.8729	0.0417	164.7818
Jacobi	1.9862	0.0138	144.2161
SSOR	1.0000	0.0471	21.2351
IP	4.6065	0.0907	50.7883

Table I shows that the SSOR preconditioner has by far the best condition number but our new method (Incomplete Poisson, referred to as IP in Table I) seems to be competitive and is significantly better than Jacobi or the pure CG method. Larger grids are evaluated with numerical experiments in Figure 3a-d. The number of iterations of CG is a well known function of the condition number [22].

C. *The IP algorithm*

In this section, we want to provide an algorithm to calculate the coefficients of the IP preconditioner in three dimensions and also account for boundary conditions. Assuming an appropriate data structure for the discretization, we show that no additional memory is necessary to compute and apply M^{-1}, which offers a matrix-free implementation of the complete Poisson solver. We assume that the boundary condition of a grid cell is stored in the data structure of the discretization. In this way, we can determine the coefficients of A by applying the known stencils for each type of cell, hence we can calculate the coefficients of L, D^{-1}, and L^T. Again, we use the compact format to describe the non-zero elements of the i-th row of the final preconditioner M^{-1}

$$\mathrm{row}_i(M^{-1}) = (m_{z-1}, m_{y-1}, m_{x-1}, m, m_{x+1}, m_{y+1}, m_{z+1}) \tag{28}$$

A deeper analysis of the matrix products in $M^{-1} = KK^T$ and the enforcement of the sparsity pattern yield the following algorithm for the 7 coefficients

$$\begin{aligned} m_{x-1} &= 1.0/A_{x-1,\mathrm{diag}} & m_{x+1} &= 1.0/A_{x+1,\mathrm{diag}} \\ m_{y-1} &= 1.0/A_{y-1,\mathrm{diag}} & m_{y+1} &= 1.0/A_{y+1,\mathrm{diag}} \\ m_{z-1} &= 1.0/A_{z-1,\mathrm{diag}} & m_{z+1} &= 1.0/A_{z+1,\mathrm{diag}} \\ m &= 1.0 + m^2_{x-1} + m^2_{y-1} + m^2_{z-1} \end{aligned} \tag{29}$$

The notation of $A_{x-1,diag}$ means that we pick the row that corresponds to the grid cell at position $(x-1,y,z)$ and in this row we access the diagonal coefficient. For the implementation, an explicit representation of the matrix A is not necessary. All that is needed is the possibility to determine the boundary condition of all neighbor cells in order to calculate the coefficients $A_{*,diag}$. In a real discretization, it is possible to store this kind of relative neighbor information in each cell to reduce data access to one element. As soon as the coefficients are known, the IP preconditioner can be applied to $\mathbf{r}_i$, which has about the same complexity as the original matrix-vector product in the CG algorithm.

V. PARALLEL MULTI-GPU IMPLEMENTATION

A. *Decomposition*

The decomposition of the discretized domain strongly affects the communication overhead between the GPUs. Reasonable decompositions of a regular grid usually include cubes, cuboids, or slabs. For simplicity we chose to use slabs which means we subdivide our domain into equally sized parts along the z-axis and distribute the data on the GPUs. As the computation of each cell only depends on the values of its direct neighbors, each node has to exchange two layers of data with at most two other nodes.

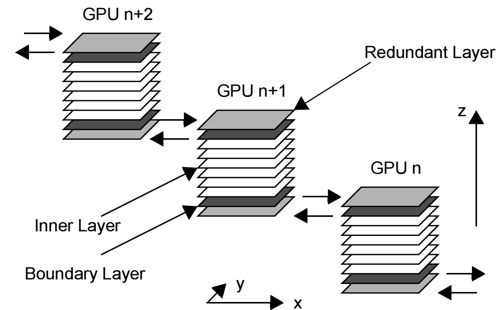

Figure 1. Data exchange of boundary layers.

According to the 7-point stencil in A and M^{-1}, the dark shaded layers of $\mathbf{d}_{i-1}$ and $\mathbf{r}_i$ (as defined in section III.C.) in Figure 1 must be transferred to the neighbor nodes in each iteration step. For the data exchange, we use redundant buffer layers for convenient access.

B. *Implementation on an NVIDIA CUDA Node*

We developed an implementation of our algorithm on a workstation with the CUDA API [19]. Our machine has the following specifications:

- 1 Intel Core i7 CPU, 2.93 GHz,
- 12 GB of main memory,
- 3 NVIDIA GTX-295 graphics cards, total of 6 GPUs, 896MB of video memory each.

In order to measure and compare our new method with state-of-the-art preconditioners we have implemented these algorithms on the GPU:

- CG (Single-, Multi-GPU, overlapping as well as non-overlapping communication),
- PCG with Jacobi preconditioner (Single-GPU),
- PCG with SSOR1 preconditioner (Single-GPU),
- PCG with SSOR2 preconditioner (Single-GPU),
- PCG with IP preconditioner (Single-, Multi-GPU, overlapping communication).

The difference in the SSOR preconditioners is the partitioning of the matrix into independent blocks. SSOR1 describes a fine-grained strategy with blocks of 128 elements, which means the matrix is partitioned into $n/128$ independent sub-matrices that are each solved by one thread block. The threads load the data elements into shared memory in parallel. Afterwards a single thread of each thread block performs the serial triangular solve step in each independent sub-matrix and writes the result back into shared memory. Subsequently, the data is written back to global memory by all threads in parallel.

The second strategy in SSOR2 is to use only one thread for the whole unmodified system. Although it is quite obvious that the performance will be poor, we take this as a reference to determine the minimum number of iteration steps because this corresponds to the exact formulation of SSOR with serial programming.

In the following, we describe our implementation of the PCG-IP method for multiple accelerators and with overlapping memory transfers. The other solvers are directly derived from this framework. The CUDA API requires a multithreaded CPU environment in order to assign tasks to the different GPUs. In order to reach a high level of concurrency, we use the concept of semaphores to handle synchronization between the data access of the threads. The CUDA framework offers a technique called *streams* to execute a kernel while a memory transfer is running, which is indispensable for our work. In a first step, we create a CPU thread for every GPU and allocate all partial vectors (see PCG algorithm) on the devices with n / #(GPUs) elements each. The simple vector operations in the PCG algorithm like addition, subtraction, and scaling can be executed in parallel by the threads without any further supervision and run just like the single-threaded version, only on smaller data sizes. The dot products, on the other hand, need a semaphore to handle synchronization. Every GPU performs the dot product with a parallel sum reduction on its part of the vectors and copies the result independently back into page-locked system memory. In order to make sure that all GPUs have finished their computation and transfer, a global barrier is needed. Afterwards the CPU sums up the partial results for the final value.

The most challenging parts of the multi-core implementation are the matrix-vector products $A\mathbf{d}_{i-1}$ and $M^{-1}\mathbf{r}_i$ with overlapping memory transfers. For the concurrency in the asynchronous data exchanges, we set up three *streams* on every GPU: one for the inner layers and two for the boundary layers (see Figure 1). The algorithm of the $A\mathbf{d}_{i-1}$ product starts with the asynchronous download of the boundary layers into page-locked system memory inside the *boundary streams*. Then we immediately trigger the computation of the inner layers in the *inner stream*, hence the memory transfers and the computation overlap. Afterwards, we continuously query the *boundary streams* if they are finished with the download. If this is the case we release a semaphore, signaling that the data is available in system memory (see Figure 2). The threads of the adjacent slabs wait for this semaphore to be released in order to copy the data from one page-locked memory area into their own container. So far, we have not been able to find a way to circumvent this additional copy operation and directly access the page-locked area of another GPU. Subsequently, the data is uploaded on the device inside the *boundary streams* while the computation of the inner layers is still running. Again, we query the *boundary streams* if they are finished with the data transfer and perform the computation of the boundary layers in a last step.

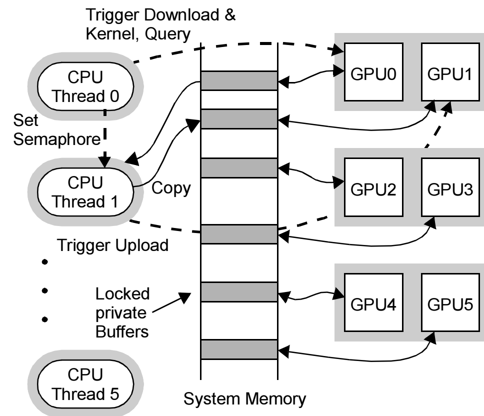

Figure 2: Overlapped kernel execution and data transfer.

The second matrix-vector product involves the IP preconditioner, which is basically the same procedure with one major exception. By looking again at the PCG algorithm we can hide the communication behind another computation:

$$\begin{aligned} \mathbf{r}_i &= \mathbf{r}_{i-1} - \alpha \mathbf{t} \\ \mathbf{p}_i &= \mathbf{p}_{i-1} + \alpha \mathbf{d}_{i-1} \\ \mathbf{h}_i &= M^{-1}\mathbf{r}_i \end{aligned} \tag{30}$$

As soon as the computation of $\mathbf{r}_i$ is finished, we trigger the download of the boundary layers in the *boundary streams*. As the CPU immediately returns from the asynchronous memory transfers, it can trigger the computation of $\mathbf{p}_i$ in the *inner stream*. During the following

matrix-vector product $M^{-1}\mathbf{r}_i$ we proceed in the same way as in the previous case with $A\mathbf{d}_{i-1}$.

VI. Performance Measurements

We subdivide the performance evaluation into two parts. First, we show results on a single GPU to analyze our IP algorithm and to compare it with GPU implementations of commonly used preconditioners. In the second part, we study the scalability using overlapped memory transfers for a multi-GPU implementation of our method.

As a basic reference, we employed a CPU implementation of the non-preconditioned CG method with OpenMP on a Core i7 at 2.93GHz with Hyperthreading and achieved a speedup of 11 with a single-GPU implementation of the IP-preconditioned variant.

We show the performance of different preconditioners on the GPU with two kinds of diagrams. At first, we plot the L_2-norm of the residual against the number of iterations to provide a platform-independent metric of the modified system when applying different preconditioners. This is shown in the left diagrams in Figure 3. In the right diagrams, we plot the L_2-norm of the residual against the computation time on the GPU, which clearly shows how the results from the first plot get distorted by the characteristics of the hardware. The SSOR2 preconditioner needs by far the lowest number of iterations to reach a certain accuracy, which is the expected behavior. This is consistent with the condition numbers from Table I and is independent from the problem size. Although the IP preconditioner needs more iterations than SSOR2, it outperforms the SSOR1 variant that uses massive partitioning and thereby loses a significant part of its theoretical advantage. The Jacobi preconditioner has almost no impact on the number of iterations, which is also consistent with the previously calculated condition numbers.

The right column in Figure 3 demonstrates how the different preconditioners behave on the graphics hardware. The right plot in Figure 3a dramatically shows how slow the serial processing of the triangular solve step of SSOR2 is. It is more than 10 times slower than the unpreconditioned CG method, which clearly disqualifies this strategy on the GPU and we omit the method in the subsequent plots. The result for 32^3 depicts the advantage of our IP preconditioner. It outperforms the CG method by a factor of 2 and is clearly faster than any other preconditioner. For growing problem sizes, the performance of the SSOR1 preconditioner severely breaks down and also becomes ineligible. In fact, among the tested preconditioners, the only method that is able to gain a speedup compared to pure CG is our new approach.

A. *Scalability*

On GPU-platforms with limited memory capacity, parallelization serves two different purposes:

- Increase the maximum problem size by domain partitioning, and
- increase processing speed.

Using the methods in this paper, problem sizes can grow proportionally to the available video memory across all graphics cards. This makes multi-GPU platforms a viable solution for many real-life applications. It is obvious, however, that the performance cannot scale arbitrarily. This is because computing costs are in the order of n^3/N, N being the number of GPUs, while communication costs are in the order of n^2. This relation is amplified by suboptimal platform architecture. In particular, direct GPU-to-GPU communication is currently not possible. Instead, data must be transferred via system memory buffers. One might expect a performance curve that converges against a certain value defined by communication costs. However, there are $2N-2$ copies from video memory to system memory (downloads), as well as $2N-2$ uploads per iteration (without preconditioner). Thus, adding more GPUs will sooner or later exhaust host capabilities, and cause the performance to drop. These effects are reflected in Table II. Here, we compare our algorithm, using overlapped communication and optionally our new preconditioner, with standard synchronized communication for various problem sizes. As expected, kernel runtimes are too short for small problem sizes (less than 256^3) so that communication costs cannot be amortized. The strength of our method shows for large grids. In all cases, however, the use of the IP preconditioner delivers a significant performance boost. It is expected that improved GPU-to-GPU communication will be standard in future GPU systems, enabling the full potential of the algorithm.

Table II: Scalability. Computing times for $L_2(\mathbf{r}) < 10^{-3}$. CG n.o. means non-overlapping Communication.

	#(GPUs)	1	2	4	6
64^3	CG n.o.	0.287s	0.820s	2.037s	3.820s
	CG	0.287s	0.723s	1.706s	3.751s
	CG IP	0.171s	0.447s	1.334s	2.672s
128^3	CG n.o.	2.540s	2.881s	4.070s	7.772s
	CG	2.540s	2.376s	3.940s	7.034s
	CG IP	1.695s	1.508s	3.108s	5.337s
256^3	CG n.o.	36.24s	21.46s	17.92s	22.38s
	CG	36.24s	17.29s	12.22s	17.52s
	CG IP	24.78s	13.67s	8.36s	13.01s
$256^2 \times 512$	CG n.o.	N/A	101.5s	67.22s	68.62s
	CG	N/A	83.92s	47.68s	48.35s
	CG IP	N/A	59.84s	36.37s	34.10s
$512^2 \times 256$	CG n.o.	N/A	N/A	183.6s	165.8s
	CG	N/A	N/A	125.6s	100.8s
	CG IP	N/A	N/A	85.92s	71.79s

VII. Conclusions

We have presented a parallel preconditioned conjugate gradient method for the Poisson equation on platforms with multiple graphics accelerators. We have shown why common preconditioners are hard to port to graphics hardware, which encouraged us to develop a new algorithm, specifically suited for the Poisson problem and for efficient GPU processing. Our heuristic approach was motivated by the Sparse Approximate Inverse algorithm to circumvent the serial processing of a triangular solve step. We were able to accelerate the CG method and to outperform classical preconditioners like SSOR on the GPU. We provided an algorithm that is fairly easy to implement and that does not require additional memory.

Furthermore, we have shown implementations of our algorithm on a multi-GPU NVIDIA workstation. We employed overlapping data transfers to minimize latency and to improve scalability.

VIII. Acknowledgments

This work was in part funded by the Deutsche Forschungsgemeinschaft under grants STR 465/14-1 (Uniboard) and SFB 716.

References

[1] M. Benzi, C. D. Meyer, and M. Tuma, "A sparse approximate inverse preconditioner for the conjugate gradient method," SIAM Journal on Scientific Computing, Volume 17, Issue 5, Sept. 1996, pp. 1135 - 1149, ISSN:1064-8275.

[2] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse matrix solvers on the GPU: conjugate gradients and multigrid," ACM Trans. Graph., 22(3):917–924, 2003. ISSN 0730-0301.

[3] R. Bridson and M. Müller-Fischer, "Fluid simulation: SIGGRAPH 2007 course notes," SIGGRAPH '07: ACM SIGGRAPH 2007 courses, ACM (2007), pp. 1–81.

[4] L. Buatois, G. Caumon, and B. Levy, "Concurrent number cruncher: a GPU implementation of a general sparse linear solver," Int. J. Parallel Emerg. Distrib. Syst., 24(3):205–223, 2009. ISSN 1744-5760.

[5] J. D. F. Cosgrove, J. C. Dias, and A. Griewank, "Approximate inverse preconditioning for sparse linear systems," Inter. J. Comp. Math., 44:91-110, 1992.

[6] J. Demmel, M. Heath, and H. van der Vorst, "Parallel numerical linear algebra," In Acta Numerica 1993. Cambridge University Press, Cambridge, UK, 1993, pp. 111-198.

[7] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha, "LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware," In SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing, page 3, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 1-59593-061-2.

[8] D. Göddeke, "GPGPU–Basic Math Tutorial," FB Mathematik, Universität Dortmund, 2005, http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html.

[9] G. H. Golub and D. P. O'Leary, "Some history of the conjugate gradient and Lanczos algorithms: 1948-1976," SIAM Review Vol. 31, No. 1 (Mar. 1989), Society for Industrial and Applied Mathematics, pp.50–102 ISSN 00361445.

[10] GPGPU. Website, 2009. http://gpgpu.org.

[11] J. D. Hall, N. A. Carr, and J. C. Hart, "Cache and bandwidth aware matrix multiplication on the GPU," 2003. UIUC Technical Report UIUCDCSR-2003-2328 (2003).

[12] M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," Journal of Research of the National Bureau of Standards, 49(6):409–436, December 1952.

[13] J. Krüger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," In SIGGRAPH '03: ACM SIGGRAPH 2003 Papers, pages 908–916, New York, NY, USA, 2003. ACM. ISBN 1-58113-709-5.

[14] E. S. Larsen and D. McAllister, "Fast matrix multiplies using graphics hardware," Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM), pages 55–55, New York, NY, USA, 2001. ACM. ISBN 1-58113-293-X.

[15] B. C. Lee, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, "Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply," ICPP '04: Proceedings of the 2004 International Conference on Parallel Processing, IEEE Computer Society, pp. 169–176.

[16] D. Luebke, M. Harris, J. Krüger, T. Purcell, N. Govindaraju, I. Buck, C. Woolley, and A. Lefohn, "GPGPU: general purpose computation on graphics hardware," SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes, ACM Press (2004).

[17] Wolfram. Mathematica. Website, 2009. http://www.wolfram.com/

[18] J. Molemaker, J. M. Cohen, S. Patel, and N. Junyong, "Low viscosity flow simulations for animation," SCA '08: Proceedings of the 2008 ACM SIGGRAPH/Eurographics symposium on Computer animation, Eurographics Association (2008), pp. 9–18.

[19] NVIDIA. CUDA. Website, 2009. http://www.nvidia.com/cuda.

[20] A. Cevahir, A. Nukada, and S. Matsuoka, "Fast conjugate gradients with multiple GPUs," Lecture Notes in Computer Science, Vol. 5544/2009, Springer Berlin / Heidelberg (May 2009), pp. 893–903 ISBN 978-3-642-01969-2.

[21] Y. Saad, "Iterative Methods for Sparse Linear Systems," Society for Industrial and Applied Mathematics; 2 edition (April 30, 2003). ISBN 0-8987-1534-2.

[22] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," Technical report, Pittsburgh, PA, USA, 1994.

[23] A. Van der Sluis and H. A. Van der Vorst, "The rate of convergence of conjugate gradients," Numer. Math., Vol. 48, No. 5, Springer-Verlag New York (1986), Inc., pp. 543–560. ISSN 0029-599X.

[24] W. A. Wiggers, V. Bakker, A. B. J. Kokkeler, and G. J. M. Smit, "Implementing the conjugate gradient algorithm on multi-core systems," In J. Nurmi, J. Takala, and O. Vainio, editors, Proceedings of the International Symposium on System-on-Chip, Tampere, pages 11–14, Piscataway, NJ, November 2007. IEEE. ISBN 1-4244-1367-2.

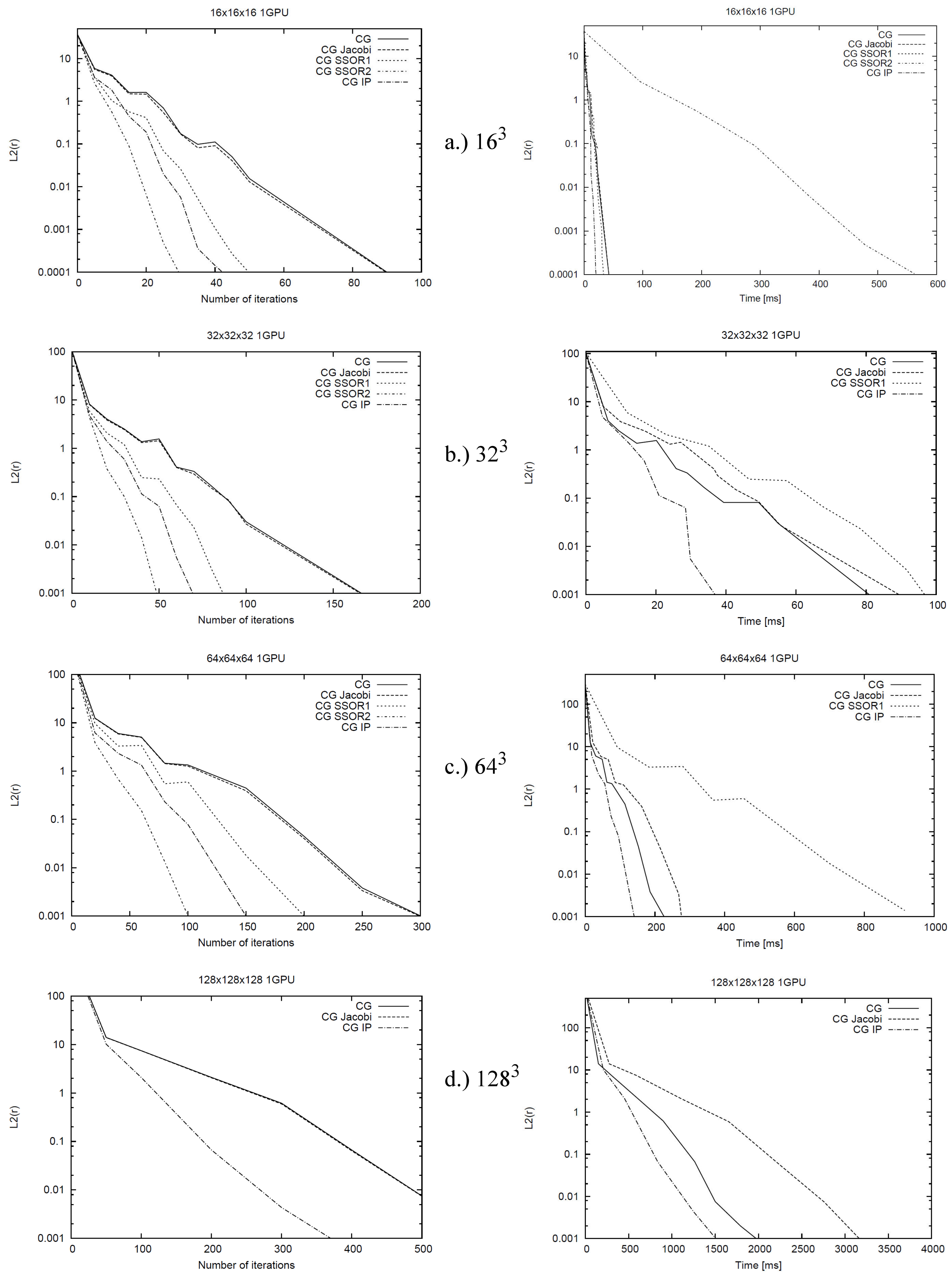

Figure 3: Results for varying problem sizes. Left: Convergence rate. Right: Processing time on GPU (see section VI).