Fast Poisson Solvers for Graphics Processing Units

Mirko Myllykoski[1], Tuomo Rossi[1], Jari Toivanen[1,2]

[1] Department of Mathematical Information Technology,
P.O. Box 35 (Agora), FI-40014 University of Jyväskylä, Finland
[2] Department of Aeronautics and Astronautics, Stanford University,
Stanford, CA 94305, USA

Abstract. Two block cyclic reduction linear system solvers are considered and implemented using the OpenCL framework. The topics of interest include a simplified scalar cyclic reduction tridiagonal system solver and the impact of increasing the radix-number of the algorithm. Both implementations are tested for the Poisson problem in two and three dimensions, using a Nvidia GTX 580 series GPU and double precision floating-point arithmetic. The numerical results indicate up to 6-fold speed increase in the case of the two-dimensional problems and up to 3-fold speed increase in the case of the three-dimensional problems when compared to equivalent CPU implementations run on a Intel Core i7 quad-core CPU.
The original publication is available at link.springer.com.
http://link.springer.com/chapter/10.1007/978-3-642-36803-5_19

1 Introduction

The linear system solvers are a very popular research topic in the field of GPU (Graphics Processing Unit, Video Card) computing. Many of these transform the original problem into a set of sub-problems which can be solved more easily. In some cases, these sub-problems are in the form of tridiagonal linear systems and the tridiagonal system solver often constitutes a significant portion of the total execution time. Conventional linear system solvers such as the LU-decomposition, also known as the Thomas method [1] when applied to a tridiagonal system, do not perform very well on a GPU because of their sequential nature. For that reason, a different kind of method called cyclic reduction [2] has become one of the most widely used methods for this purpose [3–8].

The basic idea of the cyclic reduction method can be extended to block tridiagonal systems which arise, for example, from many PDE (Partial Differential Equation) discretisations. The idea of the block cyclic reduction (BCR) was first introduced in [2]. While the formulation is numerically unstable, it can be stabilized by combining it with the Fourier analysis method [2] as was shown in [9, 10]. The first stable BCR formulation, so called Buneman's variant [11], was introduced in 1969 and generalized in [12]. Later, the idea of the partial fraction expansions was applied to the matrix rational functions occurring in the

formulas, thus leading to the discovery of a parallel variant [13]. The radix-q PSCR (Partial Solution variant of the Cyclic Reduction) method [14–17] represents a different kind of approach based on the partial solution technique [18, 19]. Excellent surveys on these kind of methods can be found in [20] and [21].

The cyclic reduction is a two-stage algorithm. The reduction stage generates a sequence of (block) tridiagonal systems by recursively eliminating (block) rows from the system and the back substitution stage solves all previously formed reduced systems in reverse order using the known rows of the solution from the previous back substitution step. Usually, the reduction is performed in such a way that all odd numbered (block) rows are eliminated, i.e., the radix-number is two. The method presented in [22] is such a method and in this paper it is called as the radix-2 BCR method. More generalized BCR methods, such as the radix-q PSCR, allow the use of higher radix-numbers.

Each radix-2 BCR reduction and back substitution step can be computed in parallel using the partial fraction expansions. However, the steps themselves must be performed sequentially. A method with a higher radix-number requires fewer steps to be taken and thus could be more suitable for parallel computation. A method analogous to the radix-2 BCR method can be easily obtained as a special case of the radix-4 PSCR method. This method reduces the systems size by a factor of four at each reduction step. Each radix-4 BCR reduction and back substitution step requires more computation than a radix-2 step, but the amount of sequential computation is reduced by a factor of two.

In this paper, the radix-2 and radix-4 BCR methods are applied to the following problem: Solve $u \in \mathbb{R}^{n_1 n_2}$ from

$$\begin{bmatrix} D & -I & & \\ -I & D & \ddots & \\ & \ddots & \ddots & -I \\ & & -I & D \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n_1} \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_{n_1} \end{bmatrix}, \tag{1}$$

where $D = \text{tridiag}\{-1, 4, -1\} \in \mathbb{R}^{n_2 \times n_2}$, when $f \in \mathbb{R}^{n_1 n_2}$ is given. It is assumed that $n_1 = 2^{k_1} - 1$ and $n_2 = 2^{k_2} - 1$ for some positive integers k_1 and k_2. This choice greatly simplifies the mathematical formulation and the implementation. The system (1) corresponds to a two-dimensional Poisson problem with Dirichlet boundary conditions posed on a rectangle.

The diagonal block can also be of the form $D = \text{tridiag}\{-I_{n_3}, \hat{D}, -I_{n_3}\} \in \mathbb{R}^{n_2 n_3 \times n_2 n_3}$, where $\hat{D} = \text{tridiag}\{-1, 6, -1\} \in \mathbb{R}^{n_3 \times n_3}$ and $n_3 = 2^{k_3} - 1$ for some positive integer k_3. In this case, the linear system (1) corresponds to a three-dimensional Poisson problem with Dirichlet boundary conditions posed in a rectangular cuboid.

The GPU implementations are compared with each other and to equivalent CPU implementations. The first objective is to find out how suitable the BCR methods are for GPU and how the radix-number effects the overall performance. The second objective is to introduce new ideas related to the tridiagonal system

solvers. In particular, it is considered how to deal with the GPU's multilevel memory architecture and its limitations.

The rest of this paper is organized as follows: The second section briefly describes the two BCR methods considered in this paper and the third section covers the key aspects of the implementation. The fourth section presents the numerical results and discussion. Finally, the conclusions are given in the fifth section.

2 Methods

2.1 Radix-2 Block Cyclic Reduction

The radix-2 BCR method can be described using the following cyclic reduction formulation described in [23]. Let $T^{(0)} = I$, $D^{(0)} = D$ and $f^{(0)} = f$. Now the reduced systems are defined, for each reduction step $r = 1, 2, \ldots, k_1 - 1$, as

$$\begin{bmatrix} D^{(r)} & -T^{(r)} & & \\ -T^{(r)} & D^{(r)} & \ddots & \\ & \ddots & \ddots & -T^{(r)} \\ & & -T^{(r)} & D^{(r)} \end{bmatrix} \begin{bmatrix} u_1^{(r)} \\ u_2^{(r)} \\ \vdots \\ u_{2^{k_1-r}-1}^{(r)} \end{bmatrix} = \begin{bmatrix} f_1^{(r)} \\ f_2^{(r)} \\ \vdots \\ f_{2^{k_1-r}-1}^{(r)} \end{bmatrix}, \tag{2}$$

where

$$\begin{aligned} T^{(r)} &= \left(T^{(r-1)}\right)^2 \left(D^{(r-1)}\right)^{-1}, \\ D^{(r)} &= D^{(r-1)} - 2\left(T^{(r-1)}\right)^2 \left(D^{(r-1)}\right)^{-1}, \\ f_i^{(r)} &= f_{2i}^{(r-1)} + T^{(r-1)} \left(D^{(r-1)}\right)^{-1} \left(f_{2i-1}^{(r-1)} + f_{2i+1}^{(r-1)}\right). \end{aligned} \tag{3}$$

These reduced systems, $r = k_1 - 1, k_1 - 2, \ldots, 0$, can be solved recursively during the back substitution stage of the algorithm by using the formula

$$u_i^{(r)} = \begin{cases} \left(D^{(r)}\right)^{-1} \left(f_i^{(r)} + T^{(r)} \left(u_{(i-1)/2}^{(r+1)} + u_{(i-1)/2+1}^{(r+1)}\right)\right), & \text{when } i \notin 2\mathbb{N}, \\ u_{i/2}^{(r+1)}, & \text{when } i \in 2\mathbb{N}, \end{cases} \tag{4}$$

where $i = 1, 2, \ldots, 2^{k_1-r} - 1$ and $u_0^{(r+1)} = u_{2^{k_1-r-1}}^{(r+1)} = 0$. Finally, $u = u^{(0)}$.

As shown in [22], if the matrices $D^{(0)}$ and $T^{(0)}$ commute, then the matrices $T^{(r)}\left(D^{(r)}\right)^{-1}$ and $\left(D^{(r)}\right)^{-1}$ can be presented using matrix polynomials and rational functions. This observation greatly improves the computational complexity of the algorithm as it preserved the sparsity properties of the coefficient matrix. Otherwise the matrices $D^{(r)}$ and $T^{(r)}$ could fill up quickly. Assuming $T^{(0)} = I$ allows the use of the partial fraction expansion technique [13] and leads to

$$T^{(r)} \left(D^{(r)}\right)^{-1} = 2^{-r} \sum_{j=1}^{2^r} (-1)^{j-1} \sin\left(\frac{2j-1}{2^{r+1}}\pi\right) \left(D - \theta(j,r) I_{n_2}\right)^{-1}, \tag{5}$$

and

$$\left(D^{(r)}\right)^{-1} = 2^{-r} \sum_{j=1}^{2^r} (D - \theta(j,r) I_{n_2})^{-1}, \tag{6}$$

where

$$\theta(j,r) = 2\cos\left(\frac{2j-1}{2^{r+1}}\pi\right). \tag{7}$$

These sum-formulations imply that each reduction and back substitution step can be carried out by first forming a large set of sub-problems, then solving these sub-problems (in parallel) and finally constructing the final result by computing collective sums over the solutions. This is the first point where some additional parallelism can be achieved and this level of parallelism is usually sufficient for contemporary multi-core CPUs.

The above described cyclic reduction formulas are well-defined (i.e. $\left(D^{(r)}\right)^{-1}$ exists for each $r = 1, 2, \ldots, k_1 - 1$) if D^{-1} exists and the coefficient matrix is strictly diagonally dominant by rows [23]. In addition, the method has been shown to be numerically stable if the smallest eigenvalue of the matrix D is at least 2 [22]. All of these conditions are fulfilled in the case of the problem (1).

The arithmetical complexity of this method is $\mathcal{O}(n_1 n_2 \log n_1)$. If the diagonal block D is block tridiagonal as discussed in the introduction, then this method can be applied recursively. In this case, the arithmetical complexity is $\mathcal{O}(n_1 n_2 n_3 \log(n_1) \log(n_2))$.

Remark 1. The above formulated partial fraction method can be actually considered to be a special case of the radix-2 PSCR method in the sense that both methods generate exactly the same sub-problems [22].

2.2 Radix-4 Block Cyclic Reduction

The formulation of the radix-4 BCR method is slightly more complicated. One approach is to start from the radix-4 PSCR method and explicitly calculate all eigenvalues and eigenvector components associated with the partial solutions. The radix-4 PSCR method can be applied to a problem with a coefficient matrix of the form

$$A_1 \otimes M_2 + M_1 \otimes A_2 + c(M_1 \otimes M_2), \tag{8}$$

where $A_1, M_1 \in \mathbb{R}^{n_1 \times n_1}$ are tridiagonal, $A_2, M_2 \in \mathbb{R}^{n_2 \times n_2}, c \in \mathbb{R}$ and $\otimes$ denotes the matrix Kronecker (tensor) product. If $A \in \mathbb{R}^{n \times n}$ and $B \in \mathbb{R}^{m \times m}$, then $A \otimes B = \{A_{i,j} B\}_{i,j=1}^{n} \in \mathbb{R}^{nm \times nm}$. The coefficient matrix in the system (1) can be expressed as

$$A \otimes I_{n_2} + I_{n_1} \otimes (D - 2I_{n_2}), \tag{9}$$

where $A = \text{tridiag}\{-1, 2, -1\} \in \mathbb{R}^{n_1 \times n_1}$.

The radix-4 PSCR method includes an initialization stage comprising generalized eigenvalue problems. Let $n_1 = 4^k - 1$ for some positive integer k. When the coefficient matrix is of the form (9), the generalized eigenvalue problems reduce to

$$\tilde{A}^{(r)} w_i^{(r)} = \lambda_i^{(r)} w_i^{(r)},\ i = 1, 2, \ldots, m_r, \tag{10}$$

where $r = 0, 1, \ldots, k-1$, $m_r = 4^{r+1} - 1$ and $\tilde{A}^{(r)} = \text{tridiag}\{-1, 2, -1\} \in \mathbb{R}^{m_r \times m_r}$.

With the assumptions mentioned above, the radix-4 PSCR solution process goes as follows: Let $f^{(0)} = f$. First, for $r = 1, 2, \ldots, k-1$, a sequence of vectors is generated by using the formula

$$f_i^{(r)} = f_{4i}^{(r-1)} + \sum_{j=1}^{m_{r-1}} (w_j^{(r-1)})_{m_{r-1}} v_{i,j}^{(r)} + \sum_{j=1}^{m_{r-1}} (w_j^{(r-1)})_1 v_{i+1,j}^{(r)}, \tag{11}$$

where $i = 1, 2, \ldots, 4^{k-r} - 1$ and the vector $v_{i,j}^{(r)}$ can be solved from

$$\left(D + (\lambda_j^{(r-1)} - 2) I_{n_2}\right) v_{i,j}^{(r)} = \sum_{s=1}^{3} (w_j^{(r-1)})_{s4^{r-1}} f_{(i-1)4+s}^{(r-1)}. \tag{12}$$

Then, for $r = k-1, k-2, \ldots, 0$, a second sequence of vectors is generated by using the formula

$$\begin{aligned} u_{4d+i}^{(r)} &= \sum_{j=1}^{m_r} (w_j^{(r)})_{i4^r} y_{d,j}^{(r)},\ i = 1, 2, 3, \\ u_{4d+4}^{(r)} &= u_{d+1}^{(r+1)}, \end{aligned} \tag{13}$$

where $d = 0, 1, \ldots, 4^{k-r} - 1$ and the vector $y_{d,j}^{(r)}$ can be solved from

$$\begin{aligned} \left(D + (\lambda_j^{(r)} - 2) I_{n_2}\right) y_{d,j}^{(r)} = \sum_{s=1}^{3} (w_j^{(r)})_{s4^r} f_{4d+s}^{(r)} + \\ (w_j^{(r)})_1 u_d^{(r+1)} + (w_j^{(r)})_{m_r} u_{d+1}^{(r+1)}. \end{aligned} \tag{14}$$

In addition, $u_0^{(r+1)} = u_{k-r-1}^{(r+1)} = 0$. Finally, $u = u^{(0)}$.

It is well-known that the matrix $\tilde{A}^{(r)}$ has the following eigenvalues and eigenvectors

$$\lambda_i^{(r)} = 2 - 2\cos\left(\frac{i\pi}{4^{r+1}}\right) \quad \text{and} \quad (w_i^{(r)})_j = \sqrt{\frac{2}{4^{r+1}}} \sin\left(\frac{ij\pi}{4^{r+1}}\right), \tag{15}$$

where $i, j = 1, 2, \ldots, m_r$. Now,

$$\begin{aligned} (w_i^{(r)})_1 &= \sqrt{2^{-2r-1}} \sin\left(i\pi/4^{r+1}\right) = (-1)^{i-1} (w_i^{(r)})_{m_r}, \\ (w_i^{(r)})_{1\cdot 4^r} &= \sqrt{2^{-2r-1}} \sin\left(i\pi/4\right) = (-1)^{i-1} (w_i^{(r)})_{3\cdot 4^r}, \\ (w_i^{(r)})_{2\cdot 4^r} &= \sqrt{2^{-2r-1}} \sin\left(i\pi/2\right). \end{aligned} \tag{16}$$

It is easy to see that $(w_j^{(r)})_{2\cdot 4^r} = 0$ when $j \in 2\mathbb{N}$ and $(w_j^{(r)})_{1\cdot 4^r} = (w_j^{(r)})_{3\cdot 4^r} = 0$ when $j \in 4\mathbb{N}$. For this reason, about one-quarter of the sub-problems required to compute the partial solutions are non-contributing and can be ignored.

Clearly each radix-4 BCR reduction and back substitution step is more computationally demanding than the corresponding radix-2 BCR step. However, the radix-2 BCR method generates a total of

$$N^2_{\text{count}}(n) = (n+1)(\log_2(n+1) - 1) + 1 \tag{17}$$

sub-problems and the radix-4 BCR method generates a total of

$$N^4_{\text{count}}(n) = (n+1)\left(\frac{3}{4}\log_2(n+1) - 1\right) + 1 \tag{18}$$

sub-problems. Thus the total number of sub-problems is reduced asymptotically by the factor

$$\lim_{n\to\infty} \frac{N^2_{\text{count}}(n)}{N^4_{\text{count}}(n)} = \frac{4}{3}. \tag{19}$$

In the case of three-dimensional problems, the ratio is even better

$$\lim_{n\to\infty} \left(\frac{N^2_{\text{count}}(n)}{N^4_{\text{count}}(n)}\right)^2 = \frac{16}{9}. \tag{20}$$

Remark 2. The above described method can be also derived by combining two radix-2 BCR reduction steps (3) into a single radix-4 BCR reduction step (11). Applying the partial fraction technique yields exactly the same sub-problems. The same procedure can be applied to the back substitution stage.

This simplified formulation can be only applied to problems with $n_1 = 4^k - 1$. However, this limitation can be easily relaxed in the following manner: Let $n_1 = 2^{\hat{k}} - 1$ for some integer $\hat{k} \geq 2$. The indexes in the reduction formula (11) are modified in such a way that $r = 1, 2, \ldots, \lceil \hat{k}/2 \rceil - 1$ and $i = 1, 2, \ldots, 2^{k-2r}$. Similarly, the indexes in the back substitution formula (13) are modified in such a way that $r = \lfloor \hat{k}/2 \rfloor - 1, \lfloor \hat{k}/2 \rfloor - 2, \ldots, 0$ and $d = 0, 1, \ldots, 2^{k-2r-2}$. If $\hat{k} \notin 2\mathbb{N}$, then it is necessary to perform one radix-2 BCR back substitution step at the radix-2 level $r = \hat{k} - 1$ in order to solve the block row $u_{2^{\hat{k}-1}}$.

The numerical experiments indicate that this method is numerically stable in the case of the Poisson problem (1). The arithmetical complexity of this method is $\mathcal{O}(n_1 n_2 \log n_1)$. If the diagonal block D is block tridiagonal as discussed in the introduction, then this method can be applied recursively. In this case, the arithmetical complexity is $\mathcal{O}(n_1 n_2 n_3 \log(n_1) \log(n_2))$.

2.3 Simplified scalar cyclic reduction

In the case of the problem (1), all tridiagonal sub-problems generated by the methods described above are of the form

$$\begin{bmatrix} d & -1 & & \\ -1 & d & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & d \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{bmatrix}, \tag{21}$$

where $d \in]2, 10[$, $v_1, \ldots, v_n, g_1, \ldots, g_n \in \mathbb{R}$ and $n = 2^k - 1$ for some positive integer k. This system can be solved with the following cyclic reduction formulas analogous to (3) and (4): Let $t^{(0)} = 1$, $d^{(0)} = d$ and $g^{(0)} = g$. Now the reduced systems are defined, for each reduction step $r = 1, 2, \ldots, k-1$, as

$$\begin{aligned} t^{(r)} &= \left(t^{(r-1)}\right)^2 / \, d^{(r-1)}, \\ d^{(r)} &= d^{(r-1)} - 2\left(t^{(r-1)}\right)^2 / \, d^{(r-1)}, \\ g_i^{(r)} &= g_{2i}^{(r-1)} + \left(t^{(r-1)} / \, d^{(r-1)}\right)\left(g_{2i-1}^{(r-1)} + g_{2i+1}^{(r-1)}\right). \end{aligned} \tag{22}$$

The solution of each reduced system, $r = k-1, k-2, \ldots, 0$, is produced recursively during the back substitution stage of the algorithm by using the formula

$$v_i^{(r)} = \begin{cases} \left(g_i^{(r)} + t^{(r)}\left(v_{(i-1)/2}^{(r+1)} + v_{(i-1)/2+1}^{(r+1)}\right)\right) / \, d^{(r)}, & \text{when } i \notin 2\mathbb{N}, \\ v_{i/2}^{(r+1)}, & \text{when } i \in 2\mathbb{N}, \end{cases} \tag{23}$$

where $i = 1, 2, \ldots, 2^{k-r} - 1$ and $v_0^{(r+1)} = v_{2^{k-r-1}}^{(r+1)} = 0$. Finally, $v = v^{(0)}$. The arithmetical complexity of this method is $\mathcal{O}(n)$.

3 Implementation

3.1 GPU Hardware

The GPU implementations are written using the OpenCL [24] framework and the OpenCL terminology is used throughout the paper. The architecture of a GPU is very different compared to a CPU. The main difference is that while a contemporary high-end consumer-level CPU may contain up to 8 cores, a modern high-end GPU contains thousands of processing elements. This means that the GPU requires a very fine-grained parallelism.

Another important difference is the memory architecture. A computing oriented GPU may include a few gigabytes of global memory (Video RAM, VRAM) which can be used to store the bulk of data. In addition, the processing elements are divided into groups called the compute units and the processing elements

belonging to the same compute unit share a fast memory area called the local memory. The effective use of this small memory area, together with a good understanding of the other underlying hardware limitations, is often the key to achieving good performance.

The GPU-side code execution begins when a special kind of subroutine called the kernel is launched. Every work-item (thread) starts from the same location in the code but each work-item is given a unique index number which makes branching possible. The work-items are divided into work groups which are then assigned to the compute units. The work-items which belong to the same work group can share a portion of the local memory.

3.2 Overall Implementation

The BCR implementations consist mostly of scalar-vector multiplications and vector-vector additions which can be implemented trivially, for example, by mapping each row-wise operation to one work-item. The large vector summations, especially during the last few reduction steps and first back substitution steps, require some additional attention. The kernels performing these summations divide the large summations into several sub-sums in order to better distribute the workload among the processing elements. The implementation employs three kernels per step approach: the first kernel generates the right-hand side vectors for the sub-problems, the second kernel solves the sub-problems and the third kernel computes the collective sums.

The implementation incorporates a simple parameter optimizer. The main application for this parametrization is to choose the optimal work group size for each kernel. Also, the kernels responsible for computing the vector sums are parametrized. The parametrization is used to choose the optimal size for each sub-sum. In addition, the parametrization is used to specify how much local memory can be used to solve a single tridiagonal sub-problem and how double precision numbers are stored into the local memory.

3.3 Previous Work on Tridiagonal System Solvers on a GPU

The GPU hardware presents many challenges to the tridiagonal system solver implementation. First, a work group can only contain a limited number work-items and the work groups cannot communicate with each other. These two limitations complicate the tasks of solving large systems. Secondly, the global memory is quite slow for scattered memory access and therefore work-items with successive index numbers should only access memory locations which are close to each other. In addition, the local memory is often divided into banks which may be subject to only one memory request at a time.

The idea of using the cyclic reduction for solving tridiagonal systems on a GPU first appeared in [3]. The cyclic reduction, the parallel cyclic reduction [25], the recursive doubling [26], and hybrid algorithms were compared with each other in [5]. All considered implementations utilize the local memory and hold the data in-place. The paper also suggested the possibility of reducing the

system size by using the cyclic reduction and the global memory in order to fit the reduced system into the local memory. The cyclic reduction and the local memory were also used in [6]. The paper introduced a clever permutation pattern which reduces the number of bank conflicts.

The idea of hybrid algorithms was taken a step further in [27]. The implementation considered consist several phases. The system is first split into multiple sub-systems using the parallel cyclic reduction and the global memory. Then, the sub-systems are solved in the local memory using the parallel cyclic reduction and the Thomas method. The optimal switching points between different stages are chosen automatically with the help of auto-tuning algorithm.

The idea of using both the global and local memory in the context of the cyclic reduction and the recursive doubling was also studied in [8]. The cyclic reduction implementation stores the right-hand side vector into the global memory and divides the system into sections. Each section is then processed separately in the local memory and then the intermediate result are merged back into the global memory. Additional work was also done in [4, 7, 28, 29].

3.4 Tridiagonal System Solver Implementation

When the coefficient matrix is a symmetric Toeplitz matrix like in (21), using the simplified cyclic reduction method is probably the most suitable algorithm for solving the tridiagonal sub-problems. The tridiagonal system solver consists of three stages and the right-hand side vector is replaced by the solution vector. One tridiagonal system is mapped to one work group and the whole solution process is performed as a single kernel launch. The implementation can be easily extended to more generalized tridiagonal systems and to cases where one tridiagonal systems is mapped to multiple work groups.

First stage. The first stage is performed only when when the system is too large to fit into the allocated local memory. It uses the global memory to store the right-hand side vector and the local memory to share odd numbered rows between work-items. The right-hand side vector is divided into sections which are the same size as the used work group. Then all sections are processed in pairs as follows: first every work-item computes one row, and then all odd numbered rows are stored into the first section, and computed rows are stored into the second section. At the next reduction step, the same procedure is repeated using the second section from each pair. This permutation pattern is reversed during the back substitution stage. Fig. 1 illustrates this process. This implementation differs from the one presented in [8].

The idea behind this segmentation and permutation pattern is to divide the right-hand side vector into independent parts which can be processed separately. In this case, these sections are processed sequentially and therefore the implementation is capable of solving systems that are too large to fit into the allocated local memory. In a more general implementation, these section can be processed in parallel using multiple work groups. The second benefit is that the rows which

belong to the same reduced system are stored close to each other in the global memory, thus allowing a more coherent global memory access pattern.

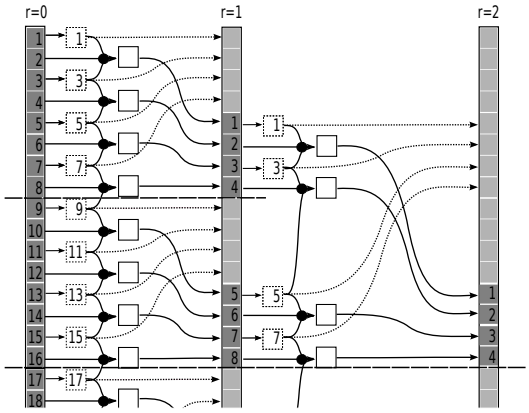

Fig. 1. The permutation pattern during the first stage of the tridiagonal system solver. The work group size is four. The numbers correspond to the row indexes. The row indexes highlighted with dotted rectangles are shared between the work-items using the local memory.

Second stage. The second stage is only performed when the number of remaining even numbered rows is greater than the used work group size. It uses a similar segmentation and permutation approach as the first stage, but the rows are processed by four sections at a time and every work-item is responsible for computing two rows. The idea is that the rows belonging to these four parts are permuted before the beginning of the reduction process in such a way that all odd numbered rows are stored into the first and third section, and all even numbered rows are stored into the second and fourth section. This permutation pattern resembles the one presented in [6]. After the reduction step is performed, the rows are permuted in such a way that all rows, which are going to be odd numbered during the next reduction step, are stored into the second section and all rows, which are going to be even numbered during the next reduction step, are stored into the fourth section. This permutation pattern is reversed during the back substitution stage. Fig. 2 illustrates this process.

The biggest advantage of this approach is that the odd and even numbered rows are located in separate sections and stored in a condensed form, thus allowing a more effective local memory access pattern when the next reduction step begins. Of course, this access pattern can still lead to bank conflicts especially when double precision arithmetic is used, as was also noted in [6]. The most straightforward solution would be to split the words and store upper and lower

bits separately but this approach was actually found to be slower. The second advantage is that the remaining right-hand side vector rows are once again divided into independent parts which can be processed separately and therefore the implementation is capable of solving systems with the number of even numbered rows higher than the used work group size.

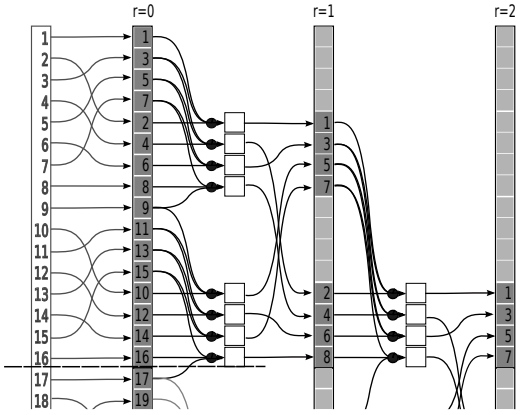

Fig. 2. The permutation pattern during the second stage of the tridiagonal system solver. The work group size is four. The numbers correspond to the row indexes.

Third stage. The last stage uses a similar row permutations as the second stage. The system is preprocessed in such a way that all even numbered rows are stored to the beginning of the memory buffer, followed by all odd numbered rows. Every work item computes at most one row. After the reduction step is performed, the rows are permuted in such a way that all rows, which are going to be even numbered during the next reduction step, are stored into the beginning of the memory buffer, followed by all rows, which are going to be odd numbered during the next reduction step. This final stage seems to be identical with the algorithm used in [6].

4 Numerical Results

The GPU tests are carried out using Nvidia GeForce GTX580 GPU with 512 processing elements (cuda cores). The CPU tests are carried out using Intel Core i7-870 2.93 GHz processor with 4 cores (8 threads). The CPU implementations are written using standard C and OpenMP framework. The CPU implementations utilize the simplified cyclic reduction, which is in this case faster than the Thomas method. All test are performed using double precision floating point arithmetic.

Fig. 3 shows results for the two-dimensional Poisson problem. Expected-line shows the expected run time difference based on (17) and (18). However, it does not take into account the memory usage and other differences. The CPU results seem to show quite constant relative run time difference between the methods. The GPU results show a much more complicated pattern. The higher than expected run time difference in the case of the small problems can be explained by the fact that the radix-4 BCR method has more parallel and less serial computation. Thus the radix-4 BCR method is better capable of taking advantage of GPU's parallel computing resources while the radix-2 BCR methods leave some of the processing elements partially unutilized.

While the radix-4 BCR method increased the amount of parallel computation, it also made it more difficult to achieve high memory throughput because the process of forming the right-hand side vectors for the sub-problems became more complicated. This is the most probable reason for the sudden drop in the performance when the problem size exceeds 1023^2. Fig. 4 shows the results for the three-dimensional Poisson problem. CPU and GPU results seem to correspond to the expectations. The sawtooth pattern is due to the modifications discussed in section 2.2.

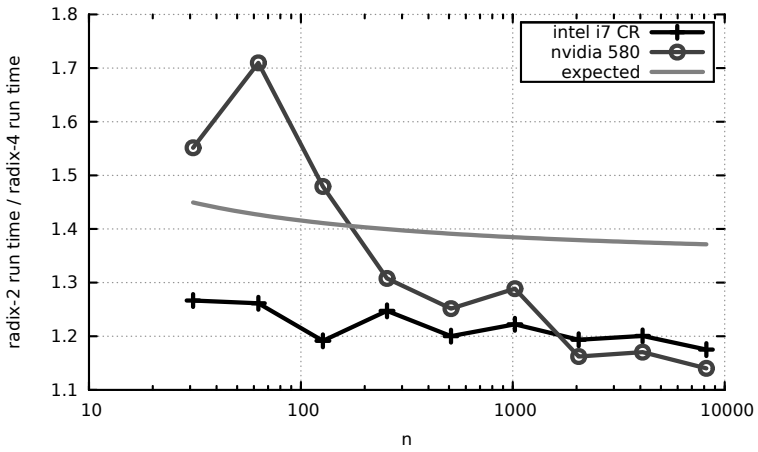

Fig. 3. Run time comparison between the radix-2 and radix-4 BCR methods, two-dimensional case, $n_1 = n_2 = n$.

Fig. 5 shows the relative run time differences between the radix-4 BCR CPU implementation and the radix-4 BCR GPU implementation. The GPU implementation is up to 6-fold faster when the transfer time between RAM and VRAM is ignored. The results for the three-dimensional GPU implementation are more modest but the GPU implementation is still up to 3-fold faster for the biggest problem.

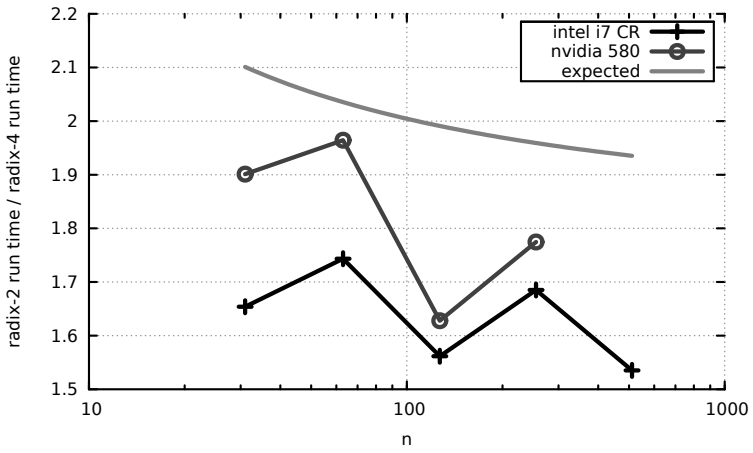

Fig. 4. Run time comparison between the radix-2 BCR and radix-4 BCR methods, three-dimensional case, $n_1 = n_2 = n_3 = n$.

5 Conclusions

This paper covered the implementation of two block cyclic reduction methods for a GPU. Special attention was given to the tridiagonal system solver. A few new ideas were introduced to improve the efficiency of the tridiagonal solver on GPUs. According to the numerical results, the block cyclic reduction algorithm seems to offer a sufficient amount of fine-grained parallelism when combined with the cyclic reduction method. The observed speed differences between the radix-2 and radix-4 methods suggests that the radix-4 version is indeed better able to take advantage of GPU's parallel computing resources.

Acknowledgements. The authors thank the reviewers for offering valuable feedback. The presentation of the paper was significantly improved thanks to their comments and suggestions. The research of the first author was supported by the Academy of Finland, grant #252549.

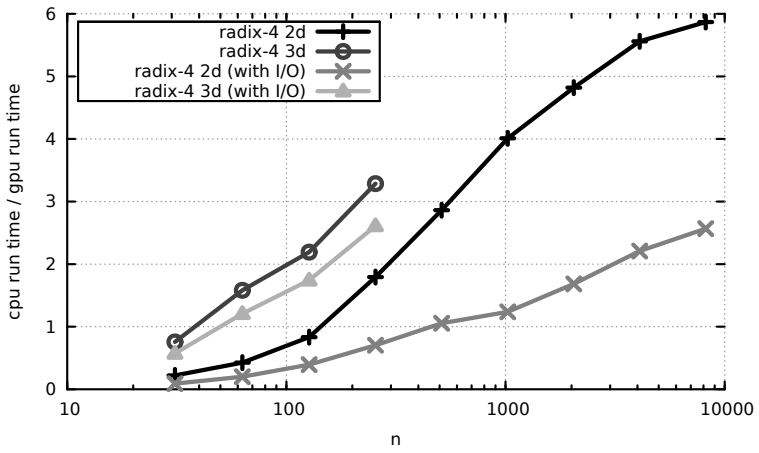

Fig. 5. Radix-4 BCR run time comparison between Intel Core i7 quad-core CPU and Nvidia GeForce GTX580 GPU, with and without initial RAM to VRAM transfer (I/O), $n_1 = n_2 = n_3 = n$.

References

1. Thomas, L.H.: Elliptic Problems in Linear Difference Equations Over a Network. Technical report, Watson Sc Comput. Lab. Rept, Columbia University, New York (1949)
2. Hockney, R. W.: A Fast Direct Solution of Poisson's Equation Using Fourier Analysis. J. Assoc. Comput. Mach. 12, 95–113 (1965)
3. Kass M., Lefohn A., Owens J. D.: Interactive Depth of Field Using Simulated Diffusion. Technical report, Pixar Animation Studios (2006)
4. Sengupta S., Harris M., Zhang Y., Owens J. D.: Scan Primitives for GPU Computing. In: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, pp. 97–106. ACM, NY, USA (2007)
5. Zhang, Y., Cohen, J., Owens, J.D.: Fast Tridiagonal Solvers on the GPU. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 127–136. ACM New York, NY, USA (2010)
6. Gödeke, D., Strzodka, R.: Cyclic Reduction Tridiagonal Solvers on GPUs Applied to Mixed Precision Multigrid. IEEE Transactions on parallel and distributed Systems (TPDS), Special issue: High Performance Computing with Accelerators 22(1), 22–32 (2011)
7. Davidson A., Owens J. D.: Register Packing for Cyclic Reduction: a Case Study. In: Proceedings of the 4th Workshop on General Purpose Processing on Graphics Processing Units, pp. 4:1–4:6. ACM, NY, USA (2011)
8. Lamas-Rodriguez J., Arguello, F., Heras, D., Boo, M.: Memory Hierarchy Optimization for Large Tridiagonal System Solvers on GPU. In: IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, pp. 87–94. IEEE (2012)
9. Hockney, R. W.: The Potential Calculation and Some Applications. Methods Comput. Phys. 9, 135–211 (1970)

10. Swarztrauber P. N.: The Method of Cyclic Reduction, Fourier Analysis and the FACR Algorithm for the Discrete Solution of Poisson's Equation on a Rectangle. SIAM Review 19, 490–501 (1977)
11. Buneman, O.: A Compact Non-Iterative Poisson Solver. Technical report 294, Stanford University Institute for Plasma Research, Stanford, CA (1969)
12. Sweet, R.A.: A Cyclic Reduction Algorithm for Solving Block Tridiagonal Systems of Arbitrary Dimension. SIAM J. Numer. Anal. 14, 706–719 (1977)
13. Sweet, R.A.: A Parallel and Vector Variant of the Cyclic Reduction Algorithm. SIAM J. Sci. Stat. Comput. 9, 761–765 (1988)
14. Vassilevski, P.: Fast Algorithm for Solving a Linear Algebraic Problem with Separable Variables. C. R. Acad. Bulgare Sci. 37, 305–308 (1984)
15. Kuznetsov, Y.A.: Numerical Methods in Subspaces. Vychislitel'nye Processy i Sistemy II, G. I. Marchuk, ed., Nauka, Moscow 37, 265–350 (1985)
16. Kuznetsov, Y.A., Rossi, T.: Fast Direct Method for Solving Algebraic Systems with Separable Symmetric Band Matrices. East-West J. Numer. Math. 4, 53–68 (1996)
17. Rossi, T.,Toivanen, J.: A Parallel Fast Direct Solver for Block Tridiagonal Systems with Separable Matrices of Arbitrary Dimension. SIAM J. Sci. Comput. 20(5), 1778–1793 (1999)
18. Banegas, A.: Fast Poisson Solvers for Problems with Sparsity. Math. Comp. 32, 441–446 (1978)
19. Kuznetsov, Y.A., Matsokin, A.M.: On Partial Solution of Systems of Linear Algebraic Equations. Sov. J. Numer. Anal. Math. Modelling 4, 453–468 (1978)
20. Bini, D.A., Meini, B.: The Cyclic Reduction Algorithm: from Poisson Equation to Stochastic Processes and Beyond. Numer. Algor. 51(1), 23–60 (2008)
21. Bialecki, B, Fairweather G. Karageorghis A.: Matrix Decomposition Algorithms for Elliptic Boundary Value Problems: a Survey. Numer. Algor. 56, 253–295 (2011)
22. Rossi, T., Toivanen, J.: A Nonstandard Cyclic Reduction Method, Its Variants and Stability. SIAM J. Matrix Anal. Appl. 20(3), 628–645 (1999)
23. Heller, D.:. Some Aspects of the Cyclic Reduction Algorithm for Block Tridiagonal Linear Systems. SIAM J. Numer. Anal. 13(4), 484–496 (1976)
24. OpenCL — The Open Standard for Parallel Programming of Heterogeneous Systems, `http://www.khronos.org/opencl/`
25. Hockney R. W., Jesshope C. R.: Parallel Computers: Architecture, Programming and algorithms. Hilger (1981)
26. Stone, H. S.: An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations. Journal of the ACM 20(1), 27–38 (1973)
27. Davidson, A., Zhang, Y., Owens, J.D.: An Auto-Tuned Method for Solving Large Tridiagonal Systems on the GPU. In: Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium. pp. 956–965. IEEE (2011)
28. Alfaro P., Igounet P., Ezzatti P.: Resolucion de Matrices Tri-diagonales Utilizando una Tarjeta Gráfica (GPU) de Escritorio. Mecanica Computacional 29(30), 2951–2967 (2010)
29. Kim H., Wu S., Chang L., Hwu W. W.: A Scalable Tridiagonal Solver for GPUs. In: Proceedings of the International Conference on Parallel Processing. pp. 444–453. IEEE (2011)