**Author(s):** Myllykoski, Mirko; Rossi, Tuomo; Toivanen, Jari

**Title:** On solving separable block tridiagonal linear systems using a GPU implementation of radix-4 PSCR method

**Year:** 2018

**Version:**

**Please cite the original version:**

Myllykoski, M., Rossi, T., & Toivanen, J. (2018). On solving separable block tridiagonal linear systems using a GPU implementation of radix-4 PSCR method. Journal of Parallel and Distributed Computing, 115(May), 56-66. https://doi.org/10.1016/j.jpdc.2018.01.004

# On solving separable block tridiagonal linear systems using a GPU implementation of radix-4 PSCR method[☆]

M. Myllykoski[a,b,1,*], T. Rossi[a], J. Toivanen[a,c,2]

[a] *Department of Mathematical Information Technology, University of Jyväskylä, P.O. Box 35 (Agora), FI-40014 University of Jyväskylä, Finland*
[b] *Department of Computing Science, Umeå University, SE-90187 Umeå, Sweden*
[c] *Department of Aeronautics & Astronautics, Stanford University, Stanford, CA 94305, USA*

**Abstract**

Partial solution variant of the cyclic reduction (PSCR) method is a direct solver that can be applied to certain types of separable block tridiagonal linear systems. Such linear systems arise, e.g., from the Poisson and the Helmholtz equations discretized with bilinear finite-elements. Furthermore, the separability of the linear system entails that the discretization domain has to be rectangular and the discretization mesh orthogonal. A generalized graphics processing unit (GPU) implementation of the PSCR method is presented. The numerical results indicate up to 24-fold speedups when compared to an equivalent CPU implementation that utilizes a single CPU core. Attained floating point performance is analyzed using roofline performance analysis model and the resulting models show that the attained floating point performance is mainly limited by the off-chip memory bandwidth and the effectiveness of a tridiagonal solver used to solve arising tridiagonal subproblems. The performance is accelerated using off-line autotuning techniques.

*Keywords:* fast direct solver, GPU computing, partial solution technique, PSCR method, roofline model, separable block tridiagonal linear system
*2010 MSC:* 35J05, 65F05, 65F50, 65N30, 65Y05

[☆]This is an accepted version of a paper published in Journal of Parallel and Distributed Computing. Please cite this article as: M. Myllykoski, T. Rossi, J. Toivanen, On solving separable block tridiagonal linear systems using a GPU implementation of radix-4 PSCR method, Journal of Parallel and Distributed Computing, 115, pp. 56–66, 2018, `https://doi.org/10.1016/j.jpdc.2018.01.004`.

[*]Corresponding author
*Email address:* `mirko.myllykoski@jyu.fi` (M. Myllykoski)

[1]The research of the first author was supported by the Academy of Finland [grant number 252549]; the Jyväskylä Doctoral Program in Computing and Mathematical Sciences; and the Foundation of Nokia Corporation.

[2]The research of the third author was supported by the Academy of Finland [grant numbers 252549, 295897].

## 1. Introduction

Separable block tridiagonal linear systems appear in many practical applications. In such a system, the coefficient matrix is presented in a separable form using Kronecker matrix tensor products. A good example is a Poisson equation with Dirichlet boundary conditions discretized in a rectangular domain using an orthogonal finite-element mesh and piecewise linear finite-elements. A similarly treated Helmholtz equation either with absorbing boundary conditions (ABC) [1, 2, 3] or a perfectly matched layer (PML) [4, 5], among others, also leads to a suitable linear system when discretized using bilinear (or trilinear) finite-elements [6]. Suitable linear systems can also be obtained using various finite difference approximations. Separable block tridiagonal systems appear as subproblems in a variety of situations. For example, a similarly discretized diffusion equation with a suitable implicit time-stepping scheme (implicit Euler being the simplest example) leads to the solution of a separable block tridiagonal linear system on each time step and suitable systems appear in image processing applications (see, e.g., [7]). In fact, the implementation described in this paper has already been used in image denoising [8].

Several effective numerical methods have been derived by employing many useful properties of the Kronecker matrix tensor forms. A comprehensive survey of these so-called matrix decomposition algorithms (MDAs) can be found in [9]. A MDA operates by reducing the linear system into a set of smaller sub-systems which are solved independently of each other. The solution of the original linear system is then obtained by reverting the reduction operation. MDAs are similar to the well-known method of separation of variables and many MDAs utilize the fast Fourier transformation method while performing the reduction operation.

Cyclic reduction methods are a well-known class of numerical algorithms that can be applied, among many other things, to tridiagonal and block tridiagonal linear systems. A traditional cyclic reduction type method (see, e.g., [10, 11, 12, 13, 14, 15]) operates in two stages. The first stage generates a sequence of sub-systems by recursively eliminating odd numbered (block) rows from the system. As a result, the size of each sub-system is approximately half of the size of the preceding sub-system. This means that the reduction factor, or the radix number as it is often called, is two. The sub-systems are solved in reverse order during the back substitution stage of the algorithm. The reduction operation often takes advantage of the properties of the coefficient matrix. For example, in certain cases, the sub-systems can be represented using matrix rational polynomials which considerably simplifies the formulas and reduces the amount of memory needed perform the computations [11, 13, 16]. A survey of the cyclic reduction methods and their applications can be found in [17].

While scalar cyclic reduction (CR) method [10] and its parallel variant (parallel cyclic reduction, PCR) [18] have become very popular methods for the solution of tridiagonal linear systems on graphics processing units (GPUs) (see, e.g., [19, 20, 21, 22, 23, 24, 25]), the block cyclic reduction type methods have been a somewhat less popular topic. Comparisons between CPU and GPU implementations of simplified radix-2 [13] and radix-4 [16] methods can be found in

[26]. The paper concluded that the block cyclic reduction type methods considered in the paper are suitable for GPU computation and that the radix-4 method is better able to utilize GPU's parallel computing resources. The presented numerical results indicate up to 6-fold speed increase for two-dimensional Poisson equations and up to 3-fold speed increase for three-dimensional Poisson equations. The implementations were compared against equivalent multi-threaded CPU implementations run on a quad-core CPU.

This paper deals with a direct method called partial solution variant of the cyclic reduction (PSCR). It is a cyclic reduction type method with MDA-type features and it can be applied to separable block tridiagonal linear systems. The initial work on the (radix-2) PSCR method was done in the 80's by Vassilevski [27, 28] and Kuznetsov [29]. The method uses so-called partial solution technique [30, 31], which can be applied effectively to a separable linear system when only a sparse set of the solution components is required and the right-hand side vector has only a few non-zero elements. The technique is a special form of MDA. A more generalized radix-q algorithm was formulated later in [32] and a parallel radix-4 CPU implementation was presented in [33]. Parallel implementations were also considered earlier in [34] and [35]. Here the radix number q means that the system size is reduced by a factor of q on each reduction step. The usual formulation of the PSCR method only distantly resembles the formulation of traditional block cyclic reduction methods. However, in certain special cases, equivalent methods can be derived using a more traditional matrix rational polynomial approach [13, 16]. If the factor matrices are tridiagonal, then the arithmetical complexity of the PSCR method is $\mathcal{O}(n_1 n_2 \log n_1)$ for $n_1 n_2 \times n_1 n_2$ problems and $\mathcal{O}(n_1 n_2 n_3 \log n_1 \log n_2)$ for $n_1 n_2 n_3 \times n_1 n_2 n_3$ problems.

This paper presents a generalized GPU implementation of the radix-4 PSCR method and in a sense this paper can be seen as a natural follow-up to [16]. The implementation can be applied to real and complex valued problems. The underlying partial differential equation (PDE) that induces the linear system can be two or three dimensional (i.e. each term in the tensor product form involves two or three factor matrices) and the factor matrices are assumed to be tridiagonal and symmetric. Certain factor matrices are assumed to be positive definite.

The implementation is tested using a Poisson equation with Dirichlet boundary conditions and a Helmholtz equation with second-order ABC. In the both cases, the domain is rectangular and the discretization is performed using an orthogonal finite-element mesh. In the case of the Poisson equation, the finite-element space consists of piecewise linear elements, and in the case of the Helmholtz equation, the finite-element space consist of bilinear elements. The numerical results are analyzed using roofline performance analysis model [36]. The model takes into account the available off-chip memory bandwidth and thus provides a very extensive picture of how effectively computational and memory resources are being utilized.

The rest of this paper is organized as follows: Section 2 introduces the reader to the CR method. Section 3 describes the Kronecker matrix tensor product, the partial solution technique, and the PSCR method. Section 4 gives

a brief introduction to GPU computing and describes the GPU implementation. Section 5 presents the numerical results and the roofline models. The final conclusions are given in Section 6.

## 2. Scalar cyclic reduction method

The PSCR method resembles the CR method in many respects. Thus, we begin by describing the CR method in a simple setting in an attempt to introduce the reader to the basic idea of cyclic reduction. Consider a tridiagonal linear system

$$\begin{bmatrix} b_1 & c_1 & & \\ a_1 & b_2 & \ddots & \\ & \ddots & \ddots & c_{2^k-2} \\ & & a_{2^k-1} & b_{2^k-1} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{2^k-1} \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_{2^k-1} \end{bmatrix}, \tag{1}$$

where $a_i, b_i, c_i, u_i, f_i \in \mathbb{K}$, $i = 1, 2, \ldots, 2^k - 1$, for some positive integer $k$. From now on, we assume that the field $\mathbb{K}$ is either $\mathbb{R}$ or $\mathbb{C}$. We focus on the equation $2j$:

$$\begin{aligned} a_{2j-1}u_{2j-2} + b_{2j-1}u_{2j-1} + c_{2j-1}u_{2j} &= f_{2j-1} \\ a_{2j}u_{2j-1} + b_{2j}u_{2j} + c_{2j}u_{2j+1} &= f_{2j} \\ a_{2j+1}u_{2j} + b_{2j+1}u_{2j+1} + c_{2j+1}u_{2j+2} &= f_{2j+1}, \end{aligned}$$

where $j = 1, 2, \ldots, 2^{k-1} - 1$, $a_1 = c_{2^k-1} = 0$ and $u_0 = u_{2^k} = 0$. We multiply the equation $2j - 1$ with $\alpha_j = -a_{2j}/b_{2j-1}$ and the equation $2j + 1$ with $\beta_j = -c_{2j}/b_{2j+1}$:

$$\begin{aligned} \alpha_j a_{2j-1}u_{2j-2} - a_{2j}u_{2j-1} + \alpha_j c_{2j-1}u_{2j} &= \alpha_j f_{2j-1} \\ a_{2j}u_{2j-1} + b_{2j}u_{2j} + c_{2j}u_{2j+1} &= f_{2j} \\ \beta_j a_{2j+1}u_{2j} - c_{2j}u_{2j+1} + \beta_j c_{2j+1}u_{2j+2} &= \beta_j f_{2j+1}. \end{aligned}$$

We can now eliminate the unknowns $u_{2j-1}$ and $u_{2j+1}$ from the equation $2j$. When we apply this procedure to all even-numbered equations in the system, we end up eliminating all odd-numbered unknowns. Furthermore, the new linear system is tridiagonal which means that we can apply this procedure recursively.

More formally, we generate a sequence of tridiagonal sub-systems as follows: Let $a_i^{(0)} = a_i$, $b_i^{(0)} = b_i$, $c_i^{(0)} = c_i$, and $f_i^{(0)} = f_i$, $i = 1, 2, \ldots, 2^k - 1$. Now, for $i = 1, 2, \ldots, k - 1$, we compute

$$\begin{aligned} \alpha_j^{(i)} &= -a_{2j}^{(i-1)}/b_{2j-1}^{(i-1)}, \ \beta_j^{(i)} = -c_{2j}^{(i-1)}/b_{2j+1}^{(i-1)}, \\ a_j^{(i)} &= \alpha_j^{(i)} a_{2j-1}^{(i-1)}, \ c_j^{(i)} = \beta_j^{(i)} c_{2j+1}^{(i-1)}, \\ b_j^{(i)} &= b_{2j}^{(i-1)} + \alpha_j^{(i)} c_{2j-1}^{(i-1)} + \beta_j^{(i)} a_{2j+1}^{(i-1)}, \\ f_j^{(i)} &= f_{2j}^{(i-1)} + \alpha_j^{(i)} f_{2j-1}^{(i-1)} + \beta_j^{(i)} f_{2j+1}^{(i-1)}, \end{aligned} \tag{2}$$

where $j = 1, 2, \ldots, 2^{k-i} - 1$. Then, for $i = k-1, k-2, \ldots, 0$, we solve each sub-system using the formula

$$u_j^{(i)} = \begin{cases} (f_j^{(i)} - a_j^{(i)} u_{(j-1)/2}^{(i+1)} - c_j^{(i)} u_{(j-1)/2+1}^{(i+1)})/b_j^{(i)}, & j \notin 2\mathbb{N}, \\ u_{j/2}^{(i+1)}, & j \in 2\mathbb{N}, \end{cases} \tag{3}$$

where $j = 1, 2, \ldots, 2^{k-i} - 1$, $a_1^{(i)} = c_{2^{k-i}-1}^{(i)} = 0$, and $u_0^{(i+1)} = u_{2^{k-i}}^{(i+1)} = 0$. Finally, $u = u^{(0)}$. The CR method can be easily generalized for linear systems of arbitrary size in which case the arithmetical complexity of the method is $\mathcal{O}(m)$, where $m$ is the number of equation in the system.

## 3. PSCR method

### *3.1. Kronecker matrix tensor product forms*

The Kronecker matrix tensor product is defined for matrices $B \in \mathbb{K}^{n_1 \times n_1}$ and $C \in \mathbb{K}^{n_2 \times n_2}$ as follows

$$B \otimes C = \begin{bmatrix} b_{1,1}C & b_{1,2}C & \ldots & b_{1,n_1}C \\ b_{2,1}C & b_{2,2}C & \ldots & b_{2,n_1}C \\ \vdots & \vdots & \ddots & \vdots \\ b_{n_1,1}C & b_{n_1,2}C & \ldots & b_{n_1,n_1}C \end{bmatrix} \in \mathbb{K}^{n_1 n_2 \times n_1 n_2}. \tag{4}$$

The product has two properties which are the basis of many MDAs: First, let $D \in \mathbb{K}^{n_1 \times n_1}$ and $E \in \mathbb{K}^{n_2 \times n_2}$. Then,

$$(B \otimes C)(D \otimes E) = BD \otimes CE \in \mathbb{K}^{n_1 n_2 \times n_1 n_2}. \tag{5}$$

Second, let $D \in \mathbb{K}^{n_1 \times n_1}$ and $E \in \mathbb{K}^{n_2 \times n_2}$. If the matrices $D$ and $E$ are nonsingular, then product matrix $D \otimes C \in \mathbb{K}^{n_1 n_2 \times n_1 n_2}$ is also nonsingular and

$$(D \otimes E)^{-1} = D^{-1} \otimes E^{-1} \in \mathbb{K}^{n_1 n_2 \times n_1 n_2}. \tag{6}$$

These two results can be derived from the definition of the Kronecker matrix tensor product.

Generally speaking, the PSCR method considered in this paper can be applied to a linear system $Au = f$, with

$$A = A_1 \otimes M_2 + M_1 \otimes A_2 + c(M_1 \otimes M_2), \tag{7}$$

where the factor matrices $A_1 \in \mathbb{K}^{n_1 \times n_1}$ and $M_1 \in \mathbb{K}^{n_1 \times n_1}$ are symmetric and tridiagonal, $A_2, M_2 \in \mathbb{K}^{n_2 \times n_2}$, and $c \in \mathbb{K}$. Thus, the coefficient matrix A is symmetric and block tridiagonal. In addition, the factor matrix $M_1$ is positive definite. Linear systems of this form usually result from the discretization of two-dimension PDEs. The method can also be used to solve three-dimensional PDEs in which case the coefficient matrix A would be of the form

$$A_1 \otimes M_2 \otimes M_3 + M_1 \otimes A_2 \otimes M_3 + M_1 \otimes M_2 \otimes A_3 + c(M_1 \otimes M_2 \otimes M_3), \tag{8}$$

where the factor matrices $A_2$ and $M_2$ are symmetric and tridiagonal, and $A_3, M_3 \in \mathbb{K}^{n_3 \times n_3}$. In addition, the factor matrices $M_1$ and $M_2$ are positive definite. From now on, we refer to linear systems with the coefficient matrices of the form (7) as two-dimensional problems and linear systems with the coefficient matrices of the form (8) as three-dimensional problems.

The PSCR method reduces a three-dimensional problem into a set of two-dimensional problems where the coefficient matrices are of the form

$$A_2 \otimes M_3 + M_2 \otimes A_3 + (c + \lambda)(M_2 \otimes M_3), \tag{9}$$

with $\lambda \in \mathbb{K}$. The PSCR can be applied recursively to solve problems with (9).

### *3.2. Overview of the algorithm*

This and the next two subsections describe the radix-q PSCR method using projection matrices similarly to [33]. We now loosely define sets $J_0, J_1, \ldots, J_k \subset \mathbb{N}$ that will indirectly determine which block rows are eliminated during each reduction step. The exact formulation of the PSCR method depends on how these sets are chosen. In the case of the radix-q PSCR method, $k = \lfloor \log_q n_1 \rfloor + 1$ and the sets fulfill the following conditions:

1. $J_0 = \{1, 2, 3, \ldots, n_1\}$, $J_k = \emptyset$.
2. $J_k \subset J_{k-1} \subset \cdots \subset J_1 \subset J_0$.
3. Let $\hat{J}_i = J_i \cup \{0, n_1 + 1\}$, $(\hat{j}_1^{(i)}, \hat{j}_2^{(i)}, \ldots)$ be the elements of the set $\hat{J}_i$ in ascending order, and

$$D_l^{(i)} = \{j \in J_{i-1} : \hat{j}_l^{(i)} < j < \hat{j}_{l+1}^{(i)}\}. \tag{10}$$

Then $\#D_l^{(i)} \leq q - 1$ for all $i = 1, 2, \ldots, k$ and $l = 1, 2, \ldots, \#J_i + 1$.

Above $\#J_i$ and $\#D_l^{(i)}$ are the cardinalities of the sets $J_i$ and $D_l^{(i)}$, respectively. The consequence of the third condition is that the rows, that are to be eliminated during a reduction step, are distributed into groups of a size of no more than $q - 1$. Examples of the sets $J_1, J_2, \ldots, J_k$ are given in Figure 1 and in the following Section 3.4.

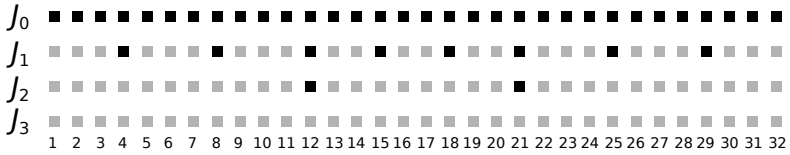


Figure 1: An example of the sets $J_0, J_1, J_2, J_3$ when $q = 4$ and $n_1 = 32$. Indexes that belong to each set are highlighted in black. The set $J_i$ implies that block rows, that correspond to indexes that belong to the set $J_{i-1} \setminus J_i$, are eliminated from the system during the $i$th reduction step.

With the sets $J_0, J_1, \ldots, J_k$, we can define projection matrices

$$\tilde{P}^{(i)} = \operatorname{diag}\{p_1^{(i)}, p_2^{(i)}, \ldots, p_{n_1}^{(i)}\} \in \mathbb{K}^{n_1},\ i = 1, 2, \ldots, k, \tag{11}$$

with

$$p_j^{(i)} = \begin{cases} 1, & j \notin J_i, \\ 0, & j \in J_i. \end{cases} \tag{12}$$

Based on these, we can define a second set of projection matrices: $P^{(i)} = \tilde{P}^{(i)} \otimes I_{n_2}$, $i = 1, 2, \ldots, k$.

Under the assumption that a projected matrix $P^{(i)}AP^{(i)}$ is nonsingular in subspace $\mathrm{Im}(P^{(i)})$ for all $i = 1, 2, \ldots, k$, the linear system $Au = f$ can be solved in two stages:

1. Let $f^{(1)} = f$. Then, for $i = 1, 2, \ldots, k-1$: Solve the vector $v^{(i)}$ from

$$P^{(i)}AP^{(i)}v^{(i)} = P^{(i)}f^{(i)} \tag{13}$$

and compute

$$f^{(i+1)} = f^{(i)} - AP^{(i)}v^{(i)}. \tag{14}$$

2. Let $u^{(k+1)} = 0$. Then, for $i = k, k-1, \ldots, 1$: Solve the vector $u^{(i)}$ from

$$P^{(i)}AP^{(i)}u^{(i)} = P^{(i)}f^{(i)} - P^{(i)}A(I - P^{(i)})u^{(i+1)} \tag{15}$$

and compute

$$(I - P^{(i)})u^{(i)} = (I - P^{(i)})u^{(i+1)}. \tag{16}$$

Finally, $u = u^{(1)}$.

The rationale behind this recursive technique becomes clear after the following observations: Due to the special block tridiagonal structure of the coefficient matrix $A$, only a sparse set of the solution components are actually required by the update formulas (14) and (16), and the right-hand side vectors in the projected systems (13) and (15) have only a few non-zero elements. In this situation the partial solution technique [30, 31] can be applied very effectively. In addition, the projected systems (13) and (15) decouple into several independent sub-systems.

### *3.3. Partial solution technique*

We will now investigate how to solve independent sub-systems in (13) and (15). Let us focus on a separable linear system $\tilde{A}v = g$ with

$$\tilde{A} = \tilde{A}_1 \otimes M_2 + \tilde{A}_1 \otimes A_2 + c(\tilde{M}_1 \otimes M_2), \tag{17}$$

where $\tilde{A}_1 \in \mathbb{K}^{m \times m}$ and $\tilde{M}_1 \in \mathbb{K}^{m \times m}$ are matching diagonal blocks from the matrices $A_1$ and $M_1$, respectively. Let us have projection matrices $\tilde{R} \in \mathbb{K}^{m \times m}$ and $\tilde{Q} \in \mathbb{K}^{m \times m}$ defining the required solution blocks and the non-zero blocks of the right-hand side vector $g$, respectively. Based on these projection matrices, we construct two additional projection matrices: $R = \tilde{R} \otimes I_{n_2}$ and $Q = \tilde{Q} \otimes I_{n_2}$.

Thus, $g \in \mathrm{Im}(Q)$ and instead of solving the whole vector $v$, we are going to solve only vector $Rv$.

The vector $Rv$ can be solved effectively by using the formula:

$$R\tilde{A}^{-1}Q = (\tilde{R}W \otimes I_{n_2})((\Lambda + cI_m) \otimes M_2 + I_m \otimes A_2)^{-1}(W^T\tilde{Q} \otimes I_{n_2}), \quad (18)$$

where the diagonal matrix $\Lambda \in \mathbb{K}^{m\times m}$ and the matrix $W \in \mathbb{K}^{m\times m}$ have the properties

$$W^T\tilde{A}_1W = \Lambda \text{ and } W^T\tilde{M}_1W = I_m. \quad (19)$$

In other words,

$$W = [w_1w_2 \dots w_m] \text{ and } \Lambda = \mathrm{diag}\{\lambda_1, \lambda_2, \dots, \lambda_m\} \quad (20)$$

contain the eigenvalues $\lambda_1, \lambda_2, \dots, \lambda_m \in \mathbb{K}$ and the $\tilde{M}_1$-orthonormal eigenvectors $w_1, w_2, \dots, w_m \in \mathbb{K}^m$, which satisfy the generalized eigenvalue problem

$$\tilde{A}_1w_j = \lambda_j\tilde{M}_1w_j. \quad (21)$$

The formula (18) follows from the properties (5) and (6).

In the case of the radix-q PSCR method, the projection matrices $\tilde{R}$ and $\tilde{Q}$ contain no more than $q+1$ non-zero blocks, which means that only $q+1$ components from each eigenvector are required to compute $Rv$. If the factor matrices $A_2$ and $M_2$ are tridiagonal, then the arithmetical complexity of computing a partial solution is $\mathcal{O}(mn_2)$, the most expensive operation being the solution of $m$ $n_2 \times n_2$ tridiagonal linear systems.

### *3.4. Explicit formulas*

This subsection gives explicit formulas in the special case when $n_1 = q^k - 1$ for some positive integer $k$. The method can be generalized for arbitrarily $n_1$ but due to lengthy formulas we limit ourselves to presenting the method in this special case. However, we emphasize that the implementation presented in this paper is applicable for general $n_1$. We begin by defining the sets $J_0, J_1, \dots, J_k \subset \mathbb{N}$ as

$$\begin{aligned} &J_0 = \{1, 2, 3, \dots, n_1\}, \\ &J_i = \left\{j \cdot q^i : j = 1, 2, \dots, q^{k-i} - 1\right\}, i = 1, 2, \dots, k-1 \text{ and} \\ &J_k = \emptyset. \end{aligned} \quad (22)$$

Clearly these sets fulfill the conditions enumerated in Section 3.2.

In summary, the radix-q PSCR method proceeds for a two-dimensional problem as follows:

1. Solve a set of generalized eigenvalue problems

$$\tilde{A}_j^{(i)}w_{j,l}^{(i)} = \lambda_{j,l}^{(i)}\tilde{M}_j^{(i)}w_{j,l}^{(i)}, \quad i = 1, 2, \dots, k, \; l = 1, 2, \dots, m_i, \quad (23)$$

where $\tilde{A}_1^{(i)}, \tilde{A}_2^{(i)}, \ldots, \tilde{A}_{q^{k-i}}^{(i)} \in \mathbb{K}^{m_i \times m_i}$ and $\tilde{M}_1^{(i)}, \tilde{M}_2^{(i)}, \ldots, \tilde{M}_{q^{k-i}}^{(i)} \in \mathbb{K}^{m_i \times m_i}$ are the non-zero diagonal blocks from the projected matrices $\tilde{P}^{(i)} A_1 \tilde{P}^{(i)}$ and $\tilde{P}^{(i)} M_1 \tilde{P}^{(i)}$, respectively, in order starting from the upper left corner, and $m_i = q^i - 1$. This step can be considered as an initialization stage. The numerical stability of the PSCR method depends largely on the properties of these generalized eigenvalue problems.

2. Let $f^{(1)} = f$. Compute the vectors $f^{(2)} \in \mathbb{K}^{(q^{k-1}-1)n_2}$, $f^{(3)} \in \mathbb{K}^{(q^{k-2}-1)n_2}$, $\ldots$, $f^{(k)} \in \mathbb{K}^{(q-1)n_2}$ by using the recursive formula

$$f_j^{(i+1)} = f_{qj}^{(i)} - \hat{T}_j^{(i)} \sum_{l=1}^{m_i} (w_{j,l}^{(i)})_{m_i} v_{j,l}^{(i)} - \check{T}_j^{(i)} \sum_{l=1}^{m_i} (w_{j+1,l}^{(i)})_1 v_{j+1,l}^{(i)} \in \mathbb{K}^{n_2}, \quad (24)$$

where

$$\begin{aligned} \hat{T}_j^{(i)} &= ((A_1)_{jq^i, jq^i-1} + c\,(M_1)_{jq^i, jq^i-1}) M_2 + (M_1)_{jq^i, jq^i-1} A_2, \\ \check{T}_j^{(i)} &= ((A_1)_{jq^i, jq^i+1} + c\,(M_1)_{jq^i, jq^i+1}) M_2 + (M_1)_{jq^i, jq^i+1} A_2. \end{aligned} \quad (25)$$

The vectors $v_{j,l}^{(i)} \in \mathbb{K}^{n_2}$ are solved from

$$\left( A_2 + (\lambda_{j,l}^{(i)} + c) M_2 \right) v_{j,l}^{(i)} = \sum_{s=1}^{q-1} (w_{j,l}^{(i)})_{sq^{i-1}} f_{(j-1)q+s}^{(i)}. \quad (26)$$

3. Compute the vectors $u^{(k)} \in \mathbb{K}^{(q-1)n_2}$, $u^{(k-1)} \in \mathbb{K}^{(q^2-1)n_2}$, $\ldots$, $u^{(1)} \in \mathbb{K}^{(q^k-1)n_2}$ by using the recursive formula

$$\begin{aligned} u_{qd+j}^{(i)} &= \sum_{l=1}^{m_i} (w_{d+1,l}^{(i)})_{jq^{i-1}} y_{d,l}^{(i)} \in \mathbb{K}^{n_2},\ j = 1, 2, \ldots, q-1, \\ u_{qd+q}^{(i)} &= u_{d+1}^{(i+1)} \in \mathbb{K}^{n_2},\ d < q^{k-i} - 1, \end{aligned} \quad (27)$$

where $d = 0, 1, \ldots, q^{k-i} - 1$. The vectors $y_{d,l}^{(i)} \in \mathbb{K}^{n_2}$ are solved from

$$\begin{aligned} \left( A_2 + (\lambda_{d+1,l}^{(i)} + c) M_2 \right) y_{d,l}^{(i)} = \sum_{s=1}^{q-1} (w_{d+1,l}^{(i)})_{sq^{i-1}} f_{qd+s}^{(i)} \\ - \hat{K}_d^{(i)} (w_{d+1,l}^{(i)})_1 u_d^{(i+1)} \\ - \check{K}_{d+1}^{(i)} (w_{d+1,l}^{(i)})_{m_i} u_{d+1}^{(i+1)}, \end{aligned} \quad (28)$$

where

$$\begin{aligned} \hat{K}_d^{(i)} &= ((A_1)_{dq^i+1, dq^i} + c\,(M_1)_{dq^i+1, dq^i}) M_2 + (M_1)_{dq^i+1, dq^i} A_2, \\ \check{K}_d^{(i)} &= ((A_1)_{dq^i-1, dq^i} + c\,(M_1)_{dq^i-1, dq^i}) M_2 + (M_1)_{dq^i-1, dq^i} A_2, \end{aligned} \quad (29)$$

$\hat{K}_0^{(i)} = \hat{K}_{q^{k-i}}^{(i)} = 0$, and $u_0^{(i+1)} = u_{q^{k-i}}^{(i+1)} = 0$. Finally, $u = u^{(1)}$.

## 4. Implementation

### *4.1. OpenCL and Nvidia's GPU hardware*

A contemporary high-end Nvidia GPU contains thousands of processing elements (CUDA cores) which are grouped into multiple computing units (streaming multiprocessors). GPU-side code execution begins when a special kind of subroutine called kernel is launched. A kernel is executed in parallel by a large set of work-items (threads) and each work-item is given a unique index number which makes branching possible. The work-items are grouped into work groups, which are then mapped to the computing units. A computing unit can execute multiple work groups concurrently but each work group is mapped to only a single computing unit.

The processing elements inside the same computing unit share certain resources such as a register file, schedulers, special function units, and caches. The schedulers can issue instructions from multiple independent instruction streams (work-items) to the same processing element. This type of over-saturated processing element occupancy is actually a desirable situation because then the instruction pipeline can be kept more easily occupied. Because multiple processing elements share the same scheduler, the code is actually executed in a synchronous manner. The hardware handles branches by going through all necessary execution paths temporally disabling those processing elements that do not contribute to the final result. Nvidia uses the term warp to describe a set of 32 work-items that are executed together in this synchronized manner.

OpenCL has two primary memory spaces:

**Global memory** is a large but relatively low bandwidth (off-chip) memory space, which can be accessed by all work-items. Attainable memory bandwidth depends largely on GPU's microarchitecture and the used access pattern. In case of older devices, a warp-coalesced access pattern, that is, a access pattern where the warp accesses memory locations that fall within the same 128 byte memory block / L1 cache line, is usually the fastest way to access the global memory. Newer devices can also benefit from warp-coalesced access but the benefits are not as substantial because only the L2 cache (32 byte cache line) can be used in most situations.

**Local memory** is a fast (on-chip) memory space which can be used when multiple work-items that belong to the same work group want to share data. The memory is divided into 32-bit (or 64-bit) memory banks organized such a way that successive 32-bit (or 64-bit) words map to successive memory banks. Each memory bank is capable of serving only one concurrent memory request. This limits the number of effective access patterns as simultaneous memory requests, that point to the same memory bank, cause a memory bank conflict and are processed sequentially.

In addition, there are two additional memory spaces called constant memory and private memory, but they are not used (explicitly) in our GPU implementation.

### *4.2. General notes*

The GPU implementation is based on the radix-4 PSCR method. The radix-4 variant was chosen because it is relatively simple to implement but still nearly optimal in the sense of arithmetical complexity [33]. Based on the numerical results presented in [26], it is also likely that a radix-4 method will outperform a radix-2 method on a GPU, especially when the problem size is small.

Our GPU implementation can be applied to problems where the factor matrices $A_1$, $A_2$, $M_1$, and $M_2$ in (7) are tridiagonal and symmetric. For three-dimensional problem, the factor matrices $A_3$ and $M_3$ in (8) are also assumed to be tridiagonal and symmetric. As mentioned earlier, the factor matrix $M_1$ (and the factor matrix $M_2$ when dealing with three-dimensional problems) is assumed to be positive definite. The field $\mathbb{K}$ can be either $\mathbb{R}$ or $\mathbb{C}$. The system size can be arbitrary as long as the GPU has enough global memory to accommodate the right-hand side vector, the factor matrices, the eigenvalues, the required eigenvector components, guiding information, and workspace buffers. Here, the term guiding information refers to data structures that are used to map work groups to different computational tasks inside the kernels.

The generalized eigenvalue problems (23) are solved on the CPU-side. This is not a major limitation because the PSCR method is usually used when one desires solve a large set of linear systems with (nearly) identical coefficient matrices. If the factor matrix $M_1$ is not diagonal, then the generalized eigenvalue problem is preprocessed with Crawford's algorithm [37] leading to an ordinary eigenvalue problem with the same eigenvalues. The eigenvalues are then solved using the LR-algorithm [38] coupled with Wilkinson's shift. The eigenvectors are solved using the inverse iteration method after the eigenvalues have been computed.

OpenCL does not have a native support for complex numbers. In the GPU implementation presented in this paper, a complex number is presented using a vector of length two. Multiplications and divisions are implemented using suitable precompiler conditionals/macros. This means that the same codebase can be used for real and complex valued problems.

Many aspects of the implementation are parametrized. For example, each kernel has its own parameter that specifies the size of the work group. If the system is complex valued, then the solver can be configured to store the real and imaginary parts separately. Similarly, the solver can be configured to store the two halves of a 64 bit double precision floating-point number as two separate 32 bit chunks. In addition, several parameters control the way the tridiagonal solver behaves. Optimal parameters that minimize the overall execution time are selected by solving the arising integer programming problems using a differential evolution [39, 40] method.

### *4.3. Upper level implementation (levels 1 and 2)*

The implementation of the PSCR method is divided into three levels: The level 3 is the tridiagonal solver, which is responsible for solving the tridiagonal subproblems in (26) and (28). The details are given in the next subsection.

The level 2 forms the right-hand side vectors for the aforementioned tridiagonal subproblems and computes the vectors (24) and (27). The level 1 is analogous to the level 2 as it performs the same operations to a three-dimensional problem. When the tridiagonal solver is excluded, the implementation consists of the following seven kernels:

`lx_stage_11` forms the right-hand side vectors for the subproblems in (26).

`lx_stage_12a` computes the vector sums in (24).

`lx_stage_y2b` helps to compute the vector sums in (24) and (27).

`lx_stage_12c` computes the vector $f^{(i+1)}$ in (24).

`lx_stage_21` forms the right-hand side vectors for the subproblems in (28).

`lx_stage_22a` computes the vector sums in (27).

`lx_stage_22c` computes the vector $u^{(i)}$ in (27).

As the levels 1 and 2 are analogous to each others, they use the same codebase. Level 1 kernels have a prefix `l1` and level 2 kernels have a prefix `l2`.

The purpose of the kernel `lx_stage_y2b` is to distribute the task of computing the vector sums in (24) and (27). Each vector sum is divided into multiple partial sums which are then evaluated in parallel by the kernels `lx_stage_12a` and `lx_stage_22a`. The same division into partial sums is then repeated recursively by the kernel `lx_stage_y2b` using a tree-like reduction pattern. The size of each partial sum is set by a parameter.

### *4.4. Tridiagonal solver (level 3)*

The tridiagonal solver used in our GPU implementation is a generalized and extended version of the CR based tridiagonal solver used in our previous paper [26]. The extended tridiagonal solver is based on the CR, PCR, and Thomas [41] methods. The CR method has been shown to be reasonably effective on GPUs due to its simplicity and parallel nature (see, e.g., [19, 20, 21, 22, 23, 24, 25]). In its basic formulation, the CR method suffers from many well-known drawbacks. The two most significant ones of them, when GPUs are concerned, are:

- The memory access pattern disperses exponentially as the method progresses. This causes problems with the global memory because warp-coalesced access is no longer possible and it causes problems with the local memory because work-items end up accessing data that is stored in the same memory bank. These problems are tackled in this paper by using different permutation schemes.

- The number of parallel tasks is reduced by a factor of two on each reduction step which leads to low processing element occupancy. This further leads to suboptimal floating point and memory performance. The situation is even worse in the complex valued case as the memory requirement is doubled when compared to the real valued case but the number of parallel tasks stays the same.

The PCR method uses the same reduction formulas as the CR method but the reduction operation is applied to every row in each reduction step. Thus, all sub-systems are the same size and the arithmetical complexity of the method is $\mathcal{O}(m \log m)$. The benefits of using the PCR method are: a more GPU-friendly memory access pattern, which does not cause additional memory bank conflicts, and high processing element occupancy. The method is widely used for solving tridiagonal systems on GPUs; see, e.g. [19, 25].

And finally, the Thomas method is a special variant of the well-known LU-decomposition method for tridiagonal matrices. The arithmetical complexity of the method is $\mathcal{O}(m)$ and it is the most effective of all the three mentioned methods. However, this sequential algorithm is not suitable for GPUs on its own. Thus, the method is often combined with the PCR method (see, e.g., [19, 22]). This is possible because a PCR reduction step splits a linear system into two independent sub-systems. After a few PCR reduction steps, the remaining sub-systems can be solved effectively using the Thomas method.

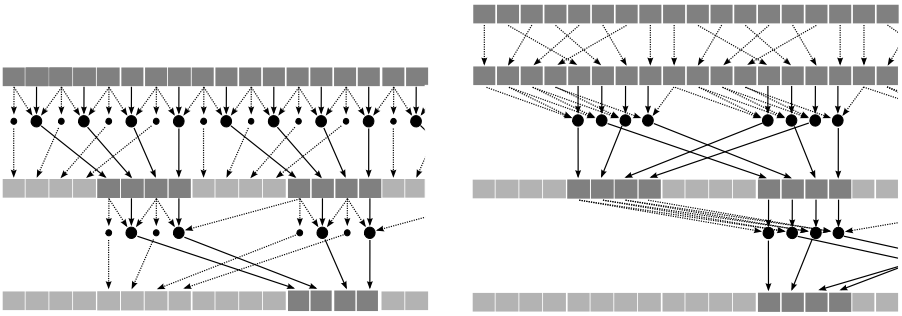

Figure 2: Examples of the permutation patterns during the stage A (on the left) and stage B (on the right) of the tridiagonal solver. In this illustration, each work group contains four work-items. In practice, the work group size is always some multiple of the warp size.

The tridiagonal solver that is used in our PSCR implementation has four main stages:

**Stage A** is used when the reduced system does not fit into the allocated local memory buffer. It this case, the coefficient matrix and the right-hand side vector are stored in the global memory and the local memory is used to share odd numbered rows between work-items. Thus each odd-numbered row needs to be read only once from the global memory. The system is divided into sections which are then processed independently of each other. The size of each section is two times the size of the work group. After a new sub-system has been computed using the CR method, each section is permuted in such a way that all odd numbered rows from the previous sub-system are stored in the upper half of the section and all rows from the new sub-system are stored in the lower half of the section. This segmentation and permutation operation is repeated after each reduction step. Figure 2 shows how the reduction stage proceeds.

The purpose of this permutation scheme is to store those rows that are needed during the next reduction step continuously in the global memory, thus avoiding the exponential memory access pattern dispersion problem mentioned earlier. Each reduction and back substitution step can be performed using multiple work groups, thus allowing a larger number of work-items to be employed per tridiagonal system.

**Stage B** is used when the reduced system fits into the allocated local memory buffer but the number of remaining even numbered rows is higher than the size of the used work group. Figure 2 shows how the reduction stage proceeds. We refer reader to [26] ("middle stage") for further details. The simplified implementation presented in [26] benefited this additional step but the benefits did not translate to the generalized implementation. The parameter optimizer kept this stage disabled during most test runs.

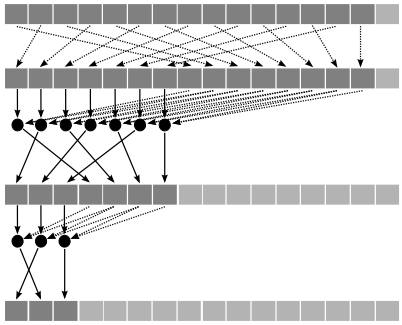

Figure 3: An example of the permutation pattern during the stage C of the tridiagonal solver. In this illustration, each work group contains eight work-items.

**Stage C** is used when the reduced system fits into the allocated local memory buffer and the number of remaining even numbered rows is at most the same as the size of the used work group. Before the first reduction step, the system is permuted in such a way that all odd numbered rows are stored in the upper half of the vector and all even numbered rows are stored in the lower half of the vector. After the new sub-system has been computed using the CR method, each vector is permuted in such a way that all computed rows, that are going to be even numbered rows during the next reduction step, are stored in the first part of the vector, followed by all computed rows, that are going to be odd numbered rows during the next reduction step. This division and permutation operation is repeated after each reduction step. This permutation pattern seems to be identical with the one used in [20].

**Stage D** solves the remaining system using a PCR-Thomas hybrid method similar to [19, 22]. This stage can be disabled completely or the Thomas-step can be skipped, if desired. In our implementation this is decided by the differential evolution optimization of the execution time.

The permutations can be implemented effectively because the coefficient matrices are stored in a vector format and the reversal of the permutation pattern during the back substitution stage is necessary only in the case of the right-hand side vectors. The tridiagonal solver presented in [26] consisted of simplified versions of the stages A, B, and C.

In total, the tridiagonal solver consists of five kernels. If it is necessary to use the global memory, then the following four kernels are used:

`l3_gen_glo_sys` is responsible for forming the coefficient matrices into the global memory.

`l3_a1` performs one stage A CR reduction step in global memory. If only one work group is used per system, then the kernel performs all remaining stage A reduction steps.

`l3_a2` performs one stage A CR back substitution step in global memory. If only one work group is used per system, then the kernel mirrors the behavior of the kernel `l3_a1`.

`l3_bcd_cpy_sys` copies the system into the local memory and performs the remaining stages B, C and D.

If there is no need to use the global memory, then only one kernel (`l3_bcd_gen_sys`) is used to performs all necessary operations.

The work group sizes, the amount of allocated local memory, switching points between different stages, the number of work groups per system, and many other properties are parametrized and, thus, optimized.

## 5. Numerical results and discussion

### *5.1. Test problems*

The first set of test problems is made out of two-dimensional and three-dimensional Poisson equations with Dirichlet boundary conditions:

$$\begin{cases} -\triangle u = f, & \text{in } \Omega, \\ u = g, & \text{on } \partial\Omega. \end{cases} \tag{30}$$

In a rectangle, the discretization using piecewise linear finite-elements on an orthogonal finite-element mesh leads the following factor matrices:

$$\begin{aligned} A_j &= \text{tridiag}\left\{ -\frac{1}{h_{j,l-1}}, \frac{h_{j,l}+h_{j,l+1}}{h_{j,l}h_{j,l+1}}, -\frac{1}{h_{j,l}} \right\} \in \mathbb{R}^{n_j \times n_j}, \\ M_j &= \text{diag}\left\{ \frac{h_{j,l}+h_{j,l+1}}{2} \right\} \in \mathbb{R}^{n_j \times n_j}, \end{aligned} \tag{31}$$

where $j = 1, 2$ or $j = 1, 2, 3$. Above, $h_{j,l}$ is the $l$th mesh step in the $j$th coordinate axis direction. In addition, $c = 0$ in (7) or (8).

The second set of test problems is made out of approximations of the Helmholtz equation

$$\begin{aligned} -\triangle u - \omega^2 u &= f, \text{ in } \mathbb{R}^d, \\ \lim_{r\to\infty} r^{(d-1)/2}\left(\frac{\partial u}{\partial r} - i\omega u\right) &= 0, \end{aligned} \tag{32}$$

where $d = 2, 3$ and $\omega$ is the wave number. The second equation of (32) is the Sommerfeld radiation condition which poses $u$ to be a radiating solution. Here $i$ denotes the imaginary unit. The unbounded domain $\mathbb{R}^d$ must be truncated to a finite one before finite element solution can be attempted. This means that we must approximate the radiation condition at the truncation boundary. Two popular ways of achieving this are a PML [4, 5] and ABC [1, 2, 3]. If the truncated domain is rectangular and the problem is discretized using bilinear (or trilinear) finite elements on an orthogonal finite-element mesh, then the resulting coefficient matrices can be presented using the Kronecker matrix tensor product in a form which is suitable for the PSCR method (see, e.g., [6]).

A second-order ABC was chosen for numerical experiments. For the two-dimensional Helmholtz equation, $c = -\omega^2$ in (7) and the factor matrices are defined as

$$A_j = \begin{bmatrix} b_{j,1} & a_{j,1} & & \\ a_{j,1} & b_{j,2} & \ddots & \\ & \ddots & \ddots & a_{j,n_j-1} \\ & & a_{j,n_j-1} & b_{j,n_j} \end{bmatrix} \in \mathbb{C}^{n_j\times n_j} \tag{33}$$

and

$$M_j = \begin{bmatrix} d_{j,1} & c_{j,1} & & \\ c_{j,1} & d_{j,2} & \ddots & \\ & \ddots & \ddots & c_{j,n_j-1} \\ & & c_{j,n_j-1} & d_{j,n_j} \end{bmatrix} \in \mathbb{C}^{n_j\times n_j}, \tag{34}$$

where

$$\begin{aligned} b_{j,l} &= \frac{h_{j,l-1}+h_{j,l}}{h_{j,l-1}h_{j,l}},\ l = 2, 3, \ldots, n_j - 1, \\ b_{j,1} &= \frac{1}{h_{j,1}} - \frac{i\omega}{2},\ b_{j,n_j} = \frac{1}{h_{j,n_j-1}} - \frac{i\omega}{2}, \\ d_{j,l} &= \frac{h_{j,l-1}+h_{j,l}}{3},\ l = 2, 3, \ldots, n_j - 1, \\ d_{j,1} &= \frac{h_{j,1}}{3} + \frac{i}{2\omega},\ d_{j,n_j} = \frac{h_{j,n_j-1}}{3} + \frac{i}{2\omega}, \\ a_{j,l} &= -\frac{1}{h_{j,l}},\ \text{ and }\ c_{j,l} = \frac{h_{j,l}}{6}. \end{aligned} \tag{35}$$

*5.2. Comparisons and general analysis of the results*

GPU experiments were carried out on a Nvidia GTX 1080 Ti GPU and an Intel i5-6600 CPU was used in CPU experiments. A single-threaded variant of the radix-4 PSCR CPU code presented in [33] was used as a CPU reference implementation. All the experiments were performed using double precision floating point arithmetic. Due to certain limitations in Nvidia's current OpenCL drivers, the GPU implementation could access only up to 4GB of global memory which limited the sizes of the numerical problems that could be considered. The GPU implementation can solve slightly larger systems than presented here but the parameter optimizer does not work properly when the required amount of global memory is close to 4GB. This is likely a memory fragmentation issue.

| $n_1$,$n_2$ | Intel i5-6600 | GTX 1080 Ti |
|---|---|---|
| 32 | 0.0001 | 0.0003 + 0.0000 |
| 64 | 0.0002 | 0.0003 + 0.0000 |
| 128 | 0.0011 | 0.0005 + 0.0001 |
| 256 | 0.0039 | 0.0005 + 0.0002 |
| 512 | 0.0200 | 0.0014 + 0.0010 |
| 1024 | 0.0829 | 0.0053 + 0.0029 |
| 2048 | 0.4005 | 0.0266 + 0.0105 |
| 4096 | 1.6845 | 0.1157 + 0.0426 |
| 8192 | 8.0182 | 0.5617 + 0.1662 |

Table 1: Execution and RAM-VRAM-RAM transfer times (in seconds) for the two-dimensional Poisson equations ($n_1 n_2$ degrees of freedom).

| $n_1$,$n_2$ | Intel i5-6600 | GTX 1080 Ti |
|---|---|---|
| 31 | 0.0001 | 0.0003 + 0.0000 |
| 63 | 0.0005 | 0.0003 + 0.0001 |
| 127 | 0.0027 | 0.0005 + 0.0001 |
| 255 | 0.0115 | 0.0011 + 0.0004 |
| 511 | 0.0593 | 0.0043 + 0.0017 |
| 1023 | 0.2292 | 0.0159 + 0.0054 |
| 2047 | 1.2619 | 0.0712 + 0.0208 |
| 4095 | 5.7642 | 0.2916 + 0.0817 |

Table 2: Execution and RAM-VRAM-RAM transfer times (in seconds) for the two-dimensional Helmholtz equations ($n_1 n_2$ degrees of freedom).

Tables 1 and 2 show the results obtained with the two-dimensional test problems. The initialization times are excluded from the tabulated execution times. The initialization time depends on the properties of the generalized eigenvalue problems (23) but in general the initialization time is one order of magnitude longer than the actual solution time. Figure 4 shows the observed speedups when the Nvidia GTX 1080 Ti GPU was used instead of one core of the Intel i5-6600 CPU. The right-hand side vector transfer times are excluded from the speedup calculations. For the Poisson equations, the GPU is up to 16 times faster and for the Helmholtz equations, the GPU is up to 20 times faster.

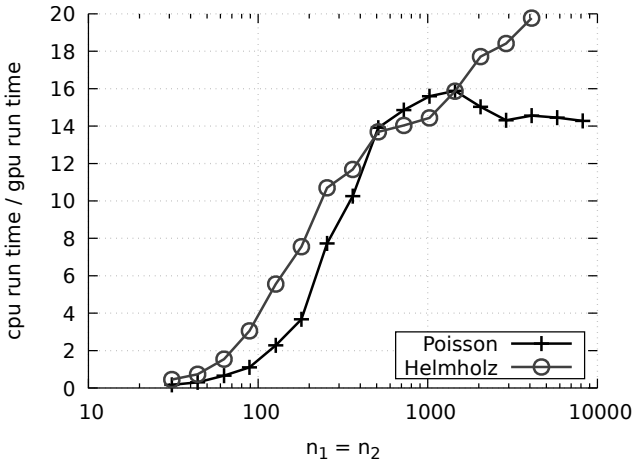


Figure 4: A execution time comparison between the Intel i5-6600 CPU and the Nvidia GTX 1080 Ti GPU for the two-dimensional problems.

The size of the problem determines the number of work-items that can be used which further determines how effectively the processing elements can be utilized. In addition, each kernel launch causes some additional overhead which has the largest impact on small problems as shown in Figure 6. These two observations explains why the speedup goes up as the problem size increases. The fact that the observed speedup flattens in the case the large Poisson equations will be discussed in Subsection 5.3.

| $n_1$,$n_2, n_3$ | Intel i5-6600 | GTX 1080 Ti |
|---|---|---|
| 31 | 0.0058 | 0.0017 + 0.0001 |
| 44 | 0.0160 | 0.0029 + 0.0004 |
| 63 | 0.0478 | 0.0047 + 0.0009 |
| 90 | 0.2422 | 0.0182 + 0.0021 |
| 127 | 0.7523 | 0.0396 + 0.0053 |
| 180 | 2.2483 | 0.1109 + 0.0158 |
| 255 | 6.5014 | 0.2734 + 0.0408 |
| 361 | 28.368 | 1.3070 + 0.1216 |
| 511 | 87.272 | 3.6158 + 0.3226 |

Table 3: Execution and RAM-VRAM-RAM transfer times (in seconds) for the three-dimensional Poisson equations ($n_1 n_2 n_3$ degrees of freedom).

Tables 3 and 4 show the corresponding results for the three-dimensional test problems. The initialization times are again excluded from the tabulated execution times. However, unlike in the case of the two-dimensional test problems, the initialization time is generally negligible when compared to the solution time. The speedup graph in Figure 5 shows that the GPU is up to 24 times faster at solving the Poisson equations and up to 18 times faster at solving the

| $n_1$,$n_2$, $n_3$ | Intel i5-6600 | GTX 1080 Ti |
|---|---|---|
| 31 | 0.0150 | 0.0030 + 0.0002 |
| 44 | 0.0432 | 0.0075 + 0.0009 |
| 63 | 0.1327 | 0.0137 + 0.0015 |
| 90 | 0.7330 | 0.0570 + 0.0040 |
| 127 | 2.1733 | 0.1358 + 0.0102 |
| 180 | 6.9103 | 0.4161 + 0.0308 |
| 255 | 20.007 | 1.0900 + 0.0819 |
| 361 | 86.519 | 5.1603 + 0.2426 |

Table 4: Execution and RAM-VRAM-RAM transfer times (in seconds) for the three-dimensional Helmholtz equations ($n_1 n_2 n_3$ degrees of freedom).

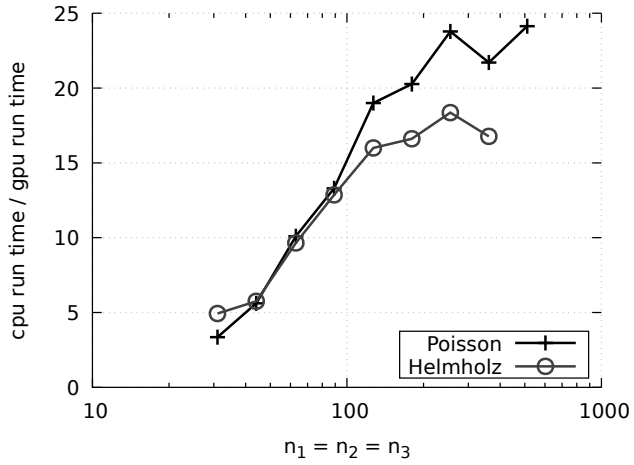


Figure 5: A execution time comparison between the Intel i5-6600 CPU and the Nvidia GTX 1080 Ti GPU for the three-dimensional test problems.

Helmholtz equations.

Figures 6 and 7 show how the execution time is distributed among different kernels on the Nvidia GTX 1080 Ti GPU. For the small two-dimensional problems, the execution time is dominated by the overhead, as expected. For the larger two-dimensional problems, the execution time is dominated by the tridiagonal solver (`l3`-kernels). Therefore, the main efforts to improve the implementation should be directed toward the tridiagonal solver. Some effort should also be directed towards the kernel `lx_stage_21`. The transition point where the tridiagonal solver begins to use the global memory (stage A) can be seen clearly at $n_1 = 1023$. For the three-dimensional problems, the overhead is negletable and the execution time is again dominated by the tridiagonal solver.

Based on numerical experiments, the GPU implementation appears to be as numerically stable as the reference CPU implementation. The parameter optimization provided some additional benefit when compared to non-optimized

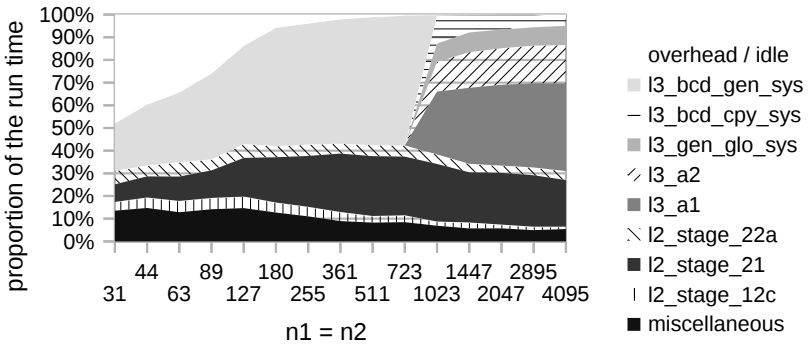


Figure 6: Execution time distribution for the two-dimensional Helmholtz equations on the Nvidia GTX 1080 Ti GPU. Miscellaneous includes all remaining kernels.

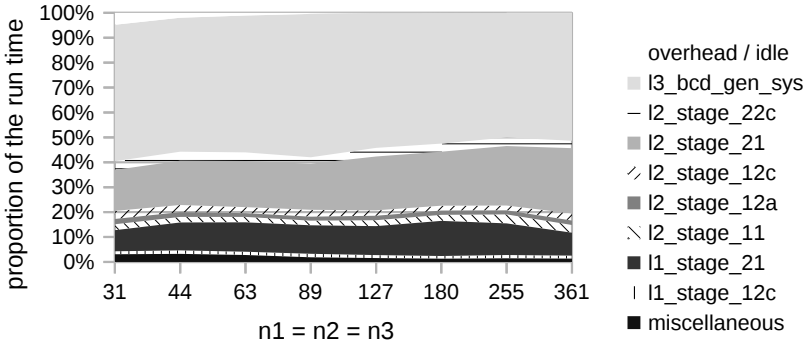


Figure 7: Execution time distribution for the three-dimensional Helmholtz equations on the Nvidia GTX 1080 Ti GPU. Miscellaneous includes all remaining kernels.

generic parameters which try to maximize processing element utilization by using large work groups and utilizing the local memory as much as possible. In case of the largest two-dimensional problems the parameter optimization lead up to 94% improvement. In other cases the improvements were more modest; average improvement being about 11%.

### *5.3. Roofline models*

The roofline model [36] is a performance analysis tool which takes into account the available off-chip memory bandwidth. The model gives a much more accurate picture than what can be obtained by simply counting the floating point operations (flop). This is especially true in the case of GPUs since the theoretical floating point performance can be remarkably high but the actual attainable floating point performance is limited in several ways, including the

global memory bandwidth. The roofline model has been previously successfully applied to GPUs; see, e.g., [42].

Basically, the application of the roofline model produces a two-dimensional scatter plot, assigning "operational intensity" to the horizontal axis, and "attained floating point performance" to the vertical axis. Here the operational intensity is defined as

$$\text{operational intensity} = \frac{\text{number of floating point operations}}{\text{number of bytes of (off-chip) memory traffic}}. \quad (36)$$

The model includes two device specific upper limits for the attainable floating point performance. The first upper limit is the theoretical peak floating point performance of the device. The second upper limit is determined by the peak (off-chip) memory bandwidth and calculated by

$$\text{peak (off-chip) memory bandwidth} \times \text{operational intensity}. \quad (37)$$

The point where these two upper bounds intersect is called device specific balance point. If the operational intensity of an algorithm is lower than the device specific balance point, then the algorithm is considered to be memory-bound; otherwise the algorithm is considered to be compute-bound. The actual attained floating point performance should be close to the upper limit specified by the operational intensity of the algorithm.

The analysis presented in this paper is based on computing an estimate for the number of floating point and memory operations on test run by test run basis. These estimates take into account the problem size and the parameters. The test run specific operational intensity and attained floating point performance were then derived from these estimates.

Figure 8 shows global memory roofline models for the Nvidia GTX 1080 Ti GPU. Based on the information provided by Nvidia, the theoretical peak double precision floating point performance was estimated to be about 332 GFlop/s (368 GFlop/s boosted) and the theoretical peak global memory bandwidth to be about 484 GB/s. This means that the device specific balance point is about 0.69 Flop/Byte, or alternatively, 5.49 flops per double precision number.

For the two-dimensional problems, the measurement points form two distinct clusters. The first cluster includes the cases in which the tridiagonal solver uses the global memory (stage A) and the second cluster includes the cases where the global memory is not used. For the two-dimensional Poisson equations, the clusters are located near (0.28 Flop/Byte, 136 GFlop/s) and (1.03 Flop/Byte, 148 GFlop/s). For the two-dimensional Helmholtz equations, the clusters are located near (0.41 Flop/Byte, 173 GFlop/s) and (1.13 Flop/Byte, 139 GFlop/s). In the case of the three-dimensional Poissons and Helmholtz equations, the clusters are located near (1.07 Flop/Byte, 235 GFlop/s) Flop/Byte and (1.32 Flop/Byte, 217 GFlop/s) Flop/Byte, respectively.

These roofline models indicate that the implementation is memory-bound for large two-dimensional problems. More careful look into the data reveals that the implementation becomes memory-bound when the tridiagonal solver

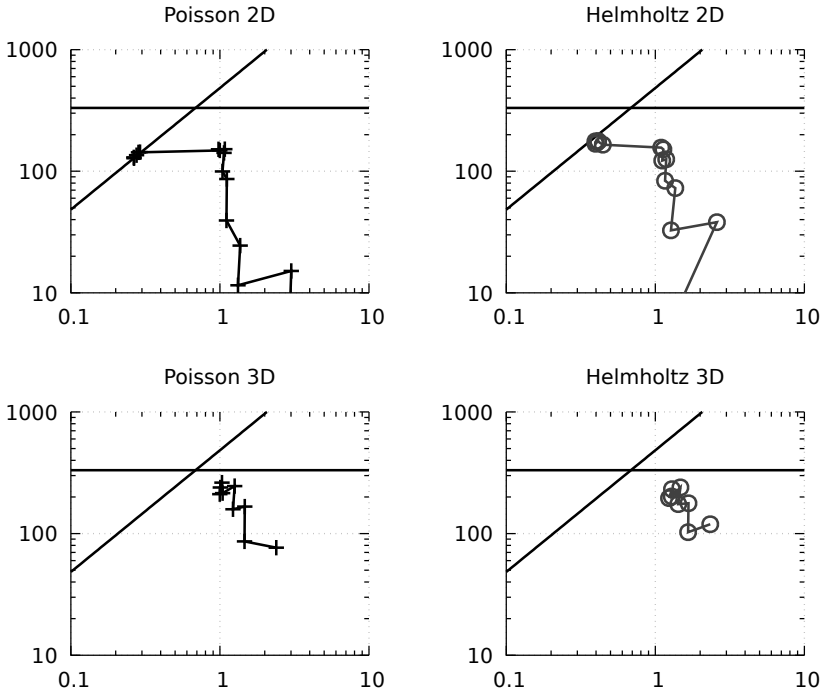


Figure 8: Roofline models for the Nvidia GTX 1080 Ti GPU. The horizontal axis shows the operational intensity [Flop/Byte] and the vertical shows the attained floating point performance [GFlop/s].

switches to using the stage A. The models also explain the observed flattening in the speedup shown in Figure 4. The model predicts that obtainable floating point performance for the first cluster of two-dimensional Poisson equations is only 484 GB/s $\times$ 0.28 Flop/Byte = 136 GFlop/s, which matches the observed performance perfectly. Thus, as the second cluster reached 148 GFlop/s performance, the obtained floating point performance is expected to drop.

In other cases the models do not completely explain the observed results as the obtained floating-point performance is slightly lower that what is predicted by the models. Several explanations were considered ranging from memory bank conflicts to potential GPU driver problems and limitations. In the end, we came to the conclusion that the most important issue to consider in this regard is the limited amount of local memory available in contemporary GPUs, which has the unfortunate side effect of severely restricting the number of work groups that can be executed simultaneously in a computing unit. This is likely to lead to a low processing element occupancy.

It appears that the CR-PCR-Thomas hybrid tridiagonal solver is unable to maintain the required processing element occupancy even though it is possible in principle by utilizing the PCR-Thomas hybrid method sufficiently. This is

due to the fact that the increase in arithmetical complexity would negate any gains made. Actually, there was little discernible benefit from using the Thomas stage at all and in most cases only the CR and PCR stages were used.

## 6. Conclusions

This paper presented a generalized GPU implementation of the radix-4 PSCR method and numerical results obtained with the implementation for four test problems. The results indicate up to 24-fold speedups when compared to an equivalent CPU implementation utilizing a single CPU core. The highest speedups were obtained with the three-dimensional Poisson equations. The numerical results indicate that the presented GPU implementation is effecient and that GPUs provide demonstrable benefit in the context of the cyclic reduction and MDA methods. The presented roofline models indicate that the performance is, for the most part, limited by the available global memory bandwidth and the effectiveness of the tridiagonal solver used to solve the arising tridiagonal subproblems. The differential evolution parameter optimization improved the average performance by 11%.

## Acknowledgements

The research of the first author was supported by the Academy of Finland [grant number 252549]; the Jyväskylä Doctoral Program in Computing and Mathematical Sciences; and the Foundation of Nokia Corporation. The research of the third author was supported by the Academy of Finland [grant numbers 252549, 295897]. The paper was revised while the first author was working at Department of Computing Science, Umeå University, Sweden. We want to thank the anonymous reviewers for their valuable feedback. We feel that the paper has been significantly improved thanks to these comments and suggestions.

## References

[1] A. Bayliss, M. Gunzburger, E. Turkel, Boundary conditions for the numerical solution of elliptic equations in exterior regions, SIAM J. Appl. Math. 42 (2) (1982) 430–451. doi:10.1137/0142032.

[2] B. Engquist, A. Majda, Absorbing boundary conditions for the numerical simulation of waves, Math. Comput. 31 (139) (1977) 629–651. doi:10.1090/S0025-5718-1977-0436612-4.

[3] D. Givoli, Non-reflecting boundary conditions, J. Comput. Phys. 94 (1) (1991) 1–29. doi:10.1016/0021-9991(91)90135-8.

[4] J. Berenger, A perfectly matched layer for the absorption of electromagnetic waves, J. Comput. Phys. 114 (2) (1994) 185–200. doi:10.1006/jcph.1994.1159.

[5] J. Berenger, Three-dimensional perfectly matched layer for the absorption of electromagnetic waves, J. Comput. Phys. 127 (2) (1996) 363–379. doi:10.1006/jcph.1996.0181.

[6] E. Heikkola, T. Rossi, J. Toivanen, Fast direct solution of the Helmholtz equation with a perfectly matched layer or an absorbing boundary condition, Inter. J. Numer. Meth. Engrg. 57 (14) (2003) 2007–2025. doi:10.1002/nme.752.

[7] M. Myllykoski, R. Glowinski, T. Kärkkäinen, T. Rossi, New augmented Lagrangian approach for $L^1$-mean curvature image denoising, SIAM J. Imaging. Sci. 8 (1) (2015) 95–125. doi:10.1137/140962164.

[8] M. Myllykoski, R. Glowinski, T. Kärkkäinen, T. Rossi, A GPU-accelerated augmented Lagrangian based $L^1$-mean curvature image denoising algorithm implementation, in: M. Gavrilova, V. Skala (Eds.), WSCG 2015 : 23rd International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision'2015 : Full Papers Proceedings, Union Agency, 2015, pp. 119–128.

[9] B. Bialecki, G. Fairweather, A. Karageorghis, Matrix decomposition algorithms for elliptic boundary value problems: a survey, Numer. Algorithms 56 (2011) 253–295. doi:10.1007/s11075-010-9384-y.

[10] R. W. Hockney, A fast direct solution of Poisson's equation using Fourier analysis, J. Assoc. Comput. Mach. 12 (1965) 95–113. doi:10.1145/321250.321259.

[11] O. Buneman, A compact non-iterative Poisson solver, Technical report 294, Institute for Plasma Research, Stanford University, Stanford, CA (1969).

[12] D. Heller, Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems, SIAM J. Numer. Anal. 13 (1976) 484–496. doi:10.1137/0713042.

[13] T. Rossi, J. Toivanen, A nonstandard cyclic reduction method, its variants and stability, SIAM J. Matrix Anal. Appl. 20 (1999) 628–645. doi:10.1137/S0895479897317053.

[14] R. A. Sweet, A cyclic reduction algorithm for solving block tridiagonal systems of arbitrary dimension, SIAM J. Numer. Anal. 14 (1977) 706–719. doi:10.1137/0714048.

[15] R. A. Sweet, A parallel and vector variant of the cyclic reduction algorithm, SIAM J. Sci. Statist. Comput. 9 (1988) 761–765. doi:10.1137/0909050.

[16] M. Myllykoski, T. Rossi, A parallel radix-4 block cyclic reduction algorithm, Numer. Linear Algebra Appl. 21 (2014) 540–556. doi:10.1002/nla.1909.

[17] D. A. Bini, B. Meini, The cyclic reduction algorithm: from Poisson equation to stochastic processes and beyond, Numer. Algorithms 51 (1) (2008) 23–60. doi:10.1007/s11075-008-9253-0.

[18] R. W. Hockney, C. R. Jesshope, Parallel computers: architecture, programming and algorithms, Bristol : Adam Hilger, 1981.

[19] A. Davidson, Y. Zhang, J. D. Owens, An auto-tuned method for solving large tridiagonal systems on the GPU, in: Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium, IEEE, 2011, pp. 956–965.

[20] D. Göddeke, R. Strzodka, Cyclic reduction tridiagonal solvers on GPUs applied to mixed precision multigrid, IEEE T. Parall. Distr., Special issue: High performance computing with accelerators 22 (1) (2011) 22–32. doi:10.1109/TPDS.2010.61.

[21] M. Kass, A. Lefohn, J. Owens, Interactive depth of field using simulated diffusion on a GPU, available as Pixar Technical Memo 06-01 (2006).

[22] H.-S. Kim, S. Wu, L. Chang, W. W. Hwu, A scalable tridiagonal solver for GPUs, 42nd International Conference on Parallel Processing (2011) 444–453doi:10.1109/ICPP.2011.41.

[23] J. Lamas-Rodriguez, F. Arguello, D. Heras, M. Boo, Memory hierarchy optimization for large tridiagonal system solvers on GPU, in: IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA), ASP-DAC '09, IEEE, IEEE Press, Piscataway, NJ, USA, 2012, pp. 87–94. doi:10.1109/ISPA.2012.20.

[24] S. Sengupta, M. Harris, Y. Zhang, J. D. Owens, Scan primitives for GPU computing, in: Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, GH '07, Eurographics Association, Aire-la-Ville, Switzerland, 2007, pp. 97–106. doi:10.2312/EGGH/EGGH07/097-106.

[25] Y. Zhang, J. Cohen, J. D. Owens, Fast tridiagonal solvers on the GPU, in: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPoPP 10, no. 1, ACM Press, 2010, pp. 127–136. doi:10.1145/1693453.1693472.

[26] M. Myllykoski, T. Rossi, J. Toivanen, Fast Poisson solvers for graphics processing units, in: P. Manninen, P. Öster (Eds.), Applied Parallel and Scientific Computing, Vol. 7782 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 265–279. doi:10.1007/978-3-642-36803-5_19.

[27] P. Vassilevski, Fast algorithm for solving a linear algebraic problem with separable variables, C.R. Acad. Bulgare Sci. 37 (1984) 305–308.

[28] P. Vassilevski, Fast algorithm for solving discrete poisson equation in a rectangle, C.R. Acad. Bulgare Sci. 38 (1985) 1311–1314.

[29] Y. A. Kuznetsov, Numerical methods in subspaces, Vychislitel'-nye Processy i Sistemy II 37 (1985) 265–350.

[30] A. Banegas, Fast Poisson solvers for problems with sparsity, Math. Comput. 32 (1978) 441–446. doi:10.2307/2006156.

[31] Y. A. Kuznetsov, A. M. Matsokin, On partial solution of systems of linear algebraic equations, Sov. J. Numer. Anal. Math. Modelling 4 (1989) 453–468. doi:10.1515/rnam.1989.4.6.453.

[32] Y. A. Kuznetsov, T. Rossi, Fast direct method for solving algebraic systems with separable symmetric band matrices, East-West J. Numer. Math. 4 (1996) 53–68.

[33] T. Rossi, J. Toivanen, A parallel fast direct solver for block tridiagonal systems with separable matrices of arbitrary dimension, SIAM J. Sci. Comput. 20 (1999) 1778–1796. doi:10.1137/S1064827597317016.

[34] A. Abakumov, Y. A. Yeremin, Y. A. Kuznetsov, Efficient fast direct method of solving Poisson's equation on a parallelepiped and its implementation in an array processor, Sov. J. Numer. Anal. Math. Model. 3 (1988) 1–20.

[35] S. Petrova, Parallel implementation of fast elliptic solver, Parallel Comput. 23 (8) (1997) 1113–1128. doi:10.1016/S0167-8191(97)00046-X.

[36] S. Williams, A. Waterman, D. Patterson, Roofline: An insightful visual performance model for multicore architectures, Commun. ACM 52 (4) (2009) 65–76. doi:10.1145/1498765.1498785.

[37] C. R. Crawford, Reduction of a band-symmetric generalized eigenvalue problem, Commun. ACM 16 (1) (1973) 41–44. doi:10.1145/361932.361943.

[38] H. Rutishauser, Solution of eigenvalue problems with the LR-transformation, U.S. Bur. Stand. Appl. Math. Ser. 49 (1958) 47–81.

[39] R. Storn, On the usage of differential evolution for function optimization, in: Fuzzy Information Processing Society, 1996. NAFIPS., 1996 Biennial Conference of the North American, 1996, pp. 519–523. doi:10.1109/NAFIPS.1996.534789.

[40] R. Storn, K. Price, Differential evolution — a simple and efficient heuristic for global optimization over continuous spaces, J. Global Optim. 11 (4) (1997) 341–359. doi:10.1023/A:1008202821328.

[41] S. D. Conte, C. de Boor, Elementary numerical analysis: an algorithmic approach, 3rd Edition, International series in pure and applied mathematics, McGraw-Hill, New York, Montreal, 1980.

[42] K.-H. Kim, K. Kim, Q.-H. Park, Performance analysis and optimization of three-dimensional FDTD on GPU using roofline model, Comput. Phys. Commun. 182 (6) (2011) 1201–1207. doi:10.1016/j.cpc.2011.01.025.