FDMAX: An Elastic Accelerator Architecture for Solving Partial Differential Equations

Jiajun Li*
Yuxuan Zhang
jiajunli@buaa.edu.cn
sy2215218@buaa.edu.cn
School of Astronautics, Beihang University
Beijing, China

Hao Zheng
hao.zheng@ucf.edu
Department of Electrical and Computer Engineering, University of Central Florida
Orlando, Florida, USA

Ke Wang
ke.wang@uncc.edu
Department of Electrical and Computer Engineering, University of North Carolina at Charlotte
Charlotte, North Carolina, USA

ABSTRACT

Partial Differential Equations (PDEs) are widely employed to describe natural phenomena in many science and engineering fields. Many PDEs do not have analytical solutions, hence, numerical methods have become prevalent for approximating PDE solutions. The most widely used numerical method is the Finite Difference Method (FDM), which requires fine grids and high-precision numerical iterations that are both compute- and memory-intensive. PDE-solving accelerators have been proposed in the literature, however, they usually focus on specific types of PDEs with rigid grid sizes which limits their broader applicability. Besides, they rarely provided insight into the optimizations of parallel computing and data accesses for solving PDEs, which hinders further improvements in performance and energy efficiency.

This paper presents FDMAX, an elastic accelerator to efficiently support FDM for different types of PDEs with any grid size. FDMAX employs a customized Processing Element (PE) array architecture that maximizes data reuse with minimized interconnection overhead. The PE array can be reconfigured to break into a set of subarrays to adapt to different grid sizes for optimal efficiency. Moreover, the PE array exploits computation and data reuse for increased performance and energy efficiency, and is reconfigurable to support a wide range of PDEs such as elliptic, parabolic, and hyperbolic equations. Evaluated on four well-known PDEs, our simulation results show that FDMAX achieves on average 1189× speedup with 1123× energy reduction over Intel Xeon CPU, and 4.9× speedup with 6.3× energy reduction over NVIDIA RTX3090 GPU, and 2.9× speedup over Alrescha, the state-of-the-art PDE-solving accelerator.

CCS CONCEPTS

• **Computer systems organization** → **Data flow architectures**; **Special purpose systems**; • **Hardware** → **Hardware accelerators**.

*Corresponding Author: Jiajun Li (jiajunli@buaa.edu.cn).

ISCA '23, June 17–21, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589083

KEYWORDS

partial differential equations, accelerator, dataflow architecture

ACM Reference Format:
Jiajun Li, Yuxuan Zhang, Hao Zheng, and Ke Wang. 2023. FDMAX: An Elastic Accelerator Architecture for Solving Partial Differential Equations. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3579371.3589083

1 INTRODUCTION

Partial Differential Equations (PDEs) are widely used to describe natural and engineering phenomena, including sound, heat, waves, and electrodynamics [13]. The solution of PDEs is a major computing problem in scientific simulation such as weather and climate prediction [34], which has become a key driver for high-performance computing.

It remains challenging to solve PDEs analytically, hence, numerical methods are prevalent to approximate the solution of PDEs. The finite difference method (FDM) [36] is the earliest yet most used numerical method that approximates PDEs by computing finite differences between discretized solutions. In practical applications, FDM usually requires fine grids and high-precision numerical iterations to obtain more accurate numerical solutions, which are both compute- and memory-intensive. Consequently, this poses a great challenge to the underlying hardware.

General-purpose processors, i.e., Central Processing Unit (CPU) and Graphics Processing Unit (GPU), can hardly deliver high-performance and energy-efficient PDE solutions as needed by a large set of applications. Therefore, specialized hardware accelerators have been proposed with substantial performance and energy efficiency gains. Specifically, Kung *et al.* [24] proposed a programmable digital accelerator for solving PDEs but they barely provided insight on the datapath and data access optimizations, leading to frequent memory accesses and excessive energy consumption. Accelerators based on processing-in-memory (PIM) technology [4] addressed the problem of data access but came with limited computing precision. Moreover, another main drawback of these accelerators is that they have limited scalability, because they usually focus on specific types of PDEs with rigid grid sizes. This limits their applicability to a broader range of applications. For example, the accelerator proposed in [32] only supports the Laplace equation with a grid size of 21×21. Its extended version [33] supports both Laplace and Poisson equations but still lacks the support for other types of equations such as Wave Equation, Heat equation, etc [13].

To this end, we study an FDM accelerator design for solving a wide range of PDEs. We first analyze the key computation pattern of PDEs and abstract it as five-point stencil computation. To efficiently support the computation pattern, we then propose a Processing Element (PE) architecture with the following merits. First, the PE is reconfigurable to support different PDEs. Second, the PE employs computation reuse to reduce costly multiplication operations. Third, we customized the datapath between PEs and between PE and on-chip buffers to minimize the data traffic. Fourth, the PE can be easily scaled to form a PE array that operates in a systolic fashion to maximize data reuse with negligible interconnection overhead.

We further propose an elastic accelerator called FDMAX based on the PE array. FDMAX is equipped with dedicated memory hierarchies to further reduce on-chip and off-chip data accesses. We observed that different PDE-solving problems prefer different scales of the PE array. Therefore, we enable the PE array to be reconfigurable to efficiently support different PDEs. Specifically, the PE array can dynamically break into several subarrays to adapt to different PDEs, which can be easily realized due to the specialized design of the PEs. The on-chip buffer structures are also reconfigurable to work together with the PE subarrays.

We built a cycle-accurate simulator to model the microarchitectural behavior of each module of FDMAX. The simulator counts the exact numbers of execution cycles, operations including multiplication/addition, and data accesses including DRAM read/write, on-chip SRAM read/write, and register file read/write. The simulator allows us to quickly explore the design space of FDMAX accelerator, and those metrics the simulator generated can be also used in the evaluation. We implement the FDMAX accelerator in RTL targeting Synopsys Generic SAED 32nm library. The experimental results show that the FDMAX with an 8 ×8 PE array can be implemented in approximately 0.99 mm^2 with power consumption at 1711.27 mW. Evaluated on four PDEs, FDMAX achieves on average 1189× speedup with 1123× energy reduction over Intel Xeon Gold 6226R CPU, and 4.9× speedup with 6.3× energy reduction over NVIDIA RTX 3090 GPU.

In summary, this paper makes the following contributions:

- **Reconfigurable PE Design.** We propose a customized PE design for the key computation pattern in FDM, which is reconfigurable to support different types of PDEs.
- **Elastic Accelerator Architecture** We propose a novel accelerator architecture design with optimizations on computation and data accesses. The accelerator is also reconfigurable to support different PDEs more efficiently.
- **Detailed Evaluation** We thoroughly evaluate the proposed accelerator by an RTL prototype and a cycle-accurate simulator, demonstrating its superior performance and energy efficiency as well as good scalability on various PDEs.

2 BACKGROUND

2.1 Partial Differential Equations

A partial differential equation is an equation that contains unknown multi-variables with their partial derivatives. Second-order PDEs are the most widely used PDEs in scientific and engineering applications. Second-order PDE containing two independent variables is an equation of the form:

$$a\frac{\partial^2 u}{\partial x^2} + b\frac{\partial^2 u}{\partial x \partial y} + c\frac{\partial^2 u}{\partial y^2} + d\frac{\partial u}{\partial x} + e\frac{\partial u}{\partial y} + fu = g \tag{1}$$

where a, b, c, d, e, f, g are functions related to x and y.

There are three types of second-order PDEs in mechanics, namely Elliptic PDE, Parabolic PDE, and Hyperbolic PDE, which depend on the value of $b^2 - 4ac$. Table 1 lists some well-known second-order equations and their applications, where ∇^2 denotes the Laplace operator [28] which is given by the sum of second partial derivatives of the function with respect to each independent variable.

Table 1: Well-known PDEs and their applications

Equation	Formula	Type	Application
Laplace Eq.	$\nabla^2 u = 0$	Elliptic	Steady heat/fluid flow
Poisson Eq.	$\nabla^2 u = b(x, y)$	Elliptic	Steady heat/fluid flow with sources or sinks
Wave Eq.	$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$	Hyperbolic	Wave motion
Heat Eq.	$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$	Parabolic	Heat conduction

2.2 Finite Difference Method

Since most PDEs do not have analytical solutions, it is prevalent to use numerical approaches to approximate the solution. FDM [36] is the most widely used numerical approach by approximating derivatives with finite differences. Both the spatial domain and time interval (if applicable) are discretized, and the value of the solution at these discrete points is approximated by solving algebraic equations containing finite differences and values from nearby points. We take the 2D Poisson equation as an example to illustrate how FDM works to solve PDE, which is given as follows:

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = b(x, y) \tag{2}$$

where $b(x, y)$ is given, and $u(x, y)$ is to be sought. The FDM converts u and b into an $M \times N$ grid of step size Δx and Δy along x and y dimension. The derivatives thus can be approximated by finite differences as follows:

$$\frac{\partial u}{\partial x}(i, j) = \frac{u_{i,j} - u_{i-1,j}}{2\Delta x}, \ \frac{\partial^2 u}{\partial x}(i, j) = \frac{u_{i-1,j} + u_{i+1,j} - 2u_{i,j}}{\Delta x^2} \tag{3}$$

The Poisson equation thus can be represented as a system of $M \times N$ equations as follows:

$$\frac{u_{i-1,j} + u_{i+1,j} - 2u_{i,j}}{\Delta x^2} + \frac{u_{i,j-1} + u_{i,j+1} - 2u_{i,j}}{\Delta y^2} = b_{i,j} \tag{4}$$
$$0 < i < M - 1, 0 < j < N - 1$$

where $u_{i,j}$ and $b_{i,j}$ are the values of u and b at grid (i, j).

As PDEs are usually specified through the boundary and initial conditions, the values of u on the border of the grid are known and those in the interior of the grid are to be sought. u and f can be flattened as a vector of length MN, i.e. $\vec{u}$ and $\vec{b}$. The system of Equations (4) can then be written as $A\vec{u} = \vec{b}$, where the stencil matrix A is an $MN \times MN$ matrix that stores the weights of u's in Equations (4). It is straightforward that $\vec{u} = A^{-1}\vec{b}$ is the solution of the equation. However, it is hard to get A^{-1} since A is a large

matrix given the large grid size. To this end, iteration methods can be used to solve the equation.

2.2.1 Jacobi Method. Jacobi method [38] is a matrix iterative method used for FDM calculations. It splits matrix A into two separate matrices, namely $A = D + R$. D is a diagonal matrix where $D_{ij} = A_{ij}$. R stores the non-diagonal elements of A, i.e. $R_{ij} = A_{ij}$ for $i \neq j$ and $R_{ii} = 0$. Hence, the solution of $\vec{u} = A^{-1}\vec{b}$, can be given by the following iterative equation:

$$u^{(k+1)} = D^{-1}(b - Ru^{(k)}) = D^{-1}b - D^{-1}Ru^{(k)} = Tu^{(k)} + c \quad (5)$$

where $u^{(k)}$ represents the u value in iteration k, $T = D^{-1}R$ is as the same size of A and is also highly sparse. Both T and c only need to be calculated once in all iterations. According to Equation (5), the u values can iteratively be calculated based on the values in the previous iteration.

Jacobi method transforms solving the system of equations into iterations that involve matrix-vector multiplications. However, T actually contains repeated values that provide data reuse opportunity, which cannot be leveraged by matrix-vector multiplication. To enable data reuse, Equation (4) can be transformed into:

$$\begin{aligned} u_{i,j}^{(k+1)} &= \frac{\Delta x^2}{2(\Delta x^2 + \Delta y^2)}(u_{i-1,j}^{(k)} + u_{i+1,j}^{(k)}) \\ &\quad + \frac{\Delta y^2}{2(\Delta x^2 + \Delta y^2)}(u_{i,j-1}^{(k)} + u_{i,j+1}^{(k)}) - \frac{\Delta x^2 \Delta y^2}{2(\Delta x^2 + \Delta y^2)} b_{i,j} \\ &= w_1(u_{i-1,j}^{(k)} + u_{i+1,j}^{(k)}) + w_2(u_{i,j-1}^{(k)} + u_{i,j+1}^{(k)}) + c_{i,j} \end{aligned} \quad (6)$$

From the above equation, the update of each $u_{i,j}$ only requires its four neighboring points $(u_{i-1,j}, u_{i+1,j}, u_{i,j-1}, u_{i,j+1})$. It significantly reduces the required memory space as it eliminated the need to store matrix T, however, it is hard to parallelize the computation on CPU or GPU.

2.2.2 Gauss-Seidel Method. In Jacobi method, the u values of the current iteration are calculated based entirely on the u values of the previous iteration. Hence, it requires storing both the u values in both the previous and current iterations. Instead of using u values of the previous iteration, Gauss-Seidel method [31] uses the latest u values available to update the values of the current iteration. Specifically, the iteration form of Gauss-Seidel method can be formulated as:

$$u_{i,j}^{(k+1)} = w_1(u_{i-1,j}^{(k+1)} + u_{i+1,j}^{(k)}) + w_2(u_{i,j-1}^{(k+1)} + u_{i,j+1}^{(k)}) + c_{i,j} \quad (7)$$

since $u_{i-1,j}^{(k+1)}$ and $u_{i,j-1}^{(k+1)}$ are already known when updating $u_{i,j}^{(k+1)}$, Gauss-Seidel method uses them to replace $u_{i-1,j}^{(k)}$ and $u_{i,j-1}^{(k)}$ in (6) to reach a faster convergence. Besides, it only needs to store the latest u values, which saves memory space. However, Gauss-Seidel method introduces data dependency since the update of each u value relies on the latest u values at its top and left points, sacrificing computing parallelism.

2.2.3 Checkerboard Method. In Checkerboard update method [39], the grid nodes are divided into two groups that are organized in checkerboard scheme. The grid nodes in each group can be in parallel thus achieving massive parallelism.

2.2.4 Hybrid Iteration Method. Hybrid iteration methods have been proposed to balance the trade-off between convergence and parallelism [18], which speed up convergence compared to Jacobi method without sacrificing all the parallelism. In this paper, we investigate a hybrid iteration method that uses the latest values from its top points. Specifically, the hybrid iteration method can be formulated as:

$$u_{i,j}^{(k+1)} = w_1(u_{i-1,j}^{(k+1)} + u_{i+1,j}^{(k)}) + w_2(u_{i,j-1}^{(k)} + u_{i,j+1}^{(k)}) + c_{i,j} \quad (8)$$

The hybrid iteration method provides a faster convergence than Jacobi method as shown in Fig. 1(b), while the update of u values in a row can be processed in parallel without data dependency.

2.2.5 Stop Condition. For Laplace and Poisson equations that do not need to find solutions over time, the iteration stops when U^k and U^{k+1} reach a stop condition. A common stop condition is that the L2 Norm of $U^k - U^{k+1}$ is smaller than a threshold, then U^{k+1} will be the solution to the equation.

3 MOTIVATION

In this section, we first identify the computation patterns in FDM for the equations listed in Table 1. We then present the motivation for designing a customized FDM accelerator.

3.1 Key Computation Patterns in FDM

From Equation (6), we know that the key computation pattern in FDM for Poisson Equation is a four-point stencil computation. Similar computation patterns can be observed in other equations such as the Laplace equation, Wave Equation, and Heat Equation. For example, the 2D Heat Equation using FDM is written as:

$$\begin{aligned} &u_{i,j}^{k+1} = r_x(u_{i-1,j}^{k} + u_{i+1,j}^{k}) + r_y(u_{i,j-1}^{k} + u_{i,j+1}^{k}) + r_c u_{i,j}^{k} \\ &where \;\; r_x = \alpha\frac{\Delta t}{\Delta x^2}, r_y = \alpha\frac{\Delta t}{\Delta y^2}, r_c = 1 - 2r_x - 2r_y \end{aligned} \quad (9)$$

It should be noted that to solve the Heat equation needs to find the solution u everywhere in x and y, and over time t. Hence, $u_{i,j}^{k}$ denotes u value at coordinate (i, j) at time step k.

Similarly, the 2D Wave equation using FDM is written as:

$$\begin{aligned} &u_{i,j}^{k+1} = r_x(u_{i-1,j}^{k} + u_{i+1,j}^{k}) + r_y(u_{i,j-1}^{k} + u_{i,j+1}^{k}) + r_c u_{i,j}^{k} - u_{i,j}^{k-1} \\ &where \;\; r_x = c^2\frac{\Delta t^2}{\Delta x^2}, r_y = c^2\frac{\Delta t^2}{\Delta y^2}, r_c = 2(1 - r_x - r_y) \end{aligned} \quad (10)$$

From the above analysis, the key computation pattern of FDM can be abstracted as a five-point stencil computation:

$$u_{i,j}^{k+1} = w_v(u_{i-1,j}^{k} + u_{i+1,j}^{k}) + w_h(u_{i,j-1}^{k} + u_{i,j+1}^{k}) + w_s u_{i,j}^{k} + b_{i,j} \quad (11)$$

where w_v and w_h denote the weights for the vertical and horizontal neighbors respectively, w_s denotes the weight of u value at the same coordinate in the previous iteration or time frame, $b_{i,j}$ denotes an offset which could be static or changing over time. Note that the u values on the right-hand side of Equation (11) could be replaced with the latest u values available if using Gauss-Seidel or hybrid update methods. In the following text, we use $U^k \in \mathcal{R}^{M \times N}$ to denote the matrix containing u values, and $B^k \in \mathcal{R}^{M \times N}$ to denote the matrix containing the offset values for a $M \times N$ grid at iteration k.

Table 2: Comparison to existing FDM accelerators and accelerators for scientific computing

Accelerator	Computing Precision	Technology	Update Method	Applications	Grid Size/Problem Size
Guo et al. [19]	Fixed 16-bit	65 nm (Analog)	-	Approximate Computing	N/A
Chen et al. [4]	Fixed 5-bit	180 nm (Analog)	Hybrid method	2D Laplace/ Poisson Eq.	Up to 128×128
Mu et al. [32]	Dynamic 4/8/12/16-bit	65 nm (Digital)	Checker-Board	2D Laplace Eq.	Fixed 21 ×21
Mu et al. [33]	Fixed 16-bit	65 nm (Digital)	Checker-Board	2D/3D Laplace/Poisson Eq.	Fixed 64 ×64(2D), 16 ×16 ×16(3D)
MemAccel [12]	Float 64-bit	15 nm (Digital)	BiCG-STAB	Systems of linear equations	Arbitrary Size
Alrescha [1]	Float 64-bit	28 nm (Digital)	PCG	Systems of linear equations	Arbitrary Size
This work	Float 32-bit	32 nm (Digital)	**Jacobi/Hybrid method**	**2D Laplace/Poisson/Heat/Wave Eq.**	**Arbitrary Size**

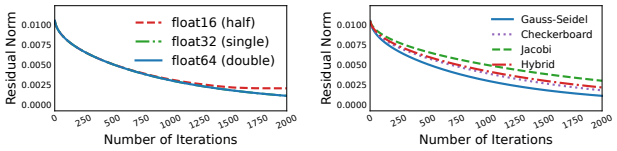
3.2 The Need for an FDM Accelerator

Nowadays more and more applications, such as weather forecasting, the prediction of economic trends, and the simulation of physical phenomena, necessitate high-performance and energy-efficient PDE solvers. For example, the basic concept of numerical weather prediction is to solve a set of PDEs that govern atmospheric motion and evolution [34, 37]. In [34], the authors reported that it requires terascale computing capability for a single PDE solver. In the simulation of physical phenomena, the PDE solvers often consume more than 95% of the execution time. For example, the simulation of combustion mainly consists of the solving of a number of PDEs [17]. The simulation of fluid dynamics [25] also consumes most of the execution time (more than 95%) in solving PDEs. Such scientific simulations are wide and important problems, which motivates the designing of customized accelerators for FDM.

3.2.1 Inefficiencies of General-Purpose Processors. Current FDM implementations on CPUs are mostly based on matrix-vector multiplication ($A\vec{u} = \vec{b}$), which usually exploits matrix processing libraries for faster speed. However, it has two disadvantages that limit performance and efficiency. First, it requires storing a large and sparse matrix which needs a large buffer memory. Second, provided that the large and sparse matrix also contains repeated values, it is hard for CPUs to exploit the data and computation reuse.

GPUs are optimized for data-parallel computations and have been widely used in applications with massive computations and memory accesses, such as deep neural networks [5]. Researchers have also implemented FDM on GPUs for acceleration [29, 30]. However, GPUs are notorious for being power-hungry and less energy efficient. For example, it consumes at least 320 J of energy to solve a 2D Poisson equation in a 128×128 grid using GPU [4].

3.2.2 Limitation of Existing FDM Accelerators. Customized FDM accelerators have been proposed and delivered substantial gains on both performance and energy efficiency. Table 2 summarized the characteristics of these accelerators. Overall, existing FDM accelerators encounter several challenges that limit their applicability. Accelerators based on processing-in-memory (PIM) suffered from limited computing precision [4]. Analog-based accelerators introduce digital-analog conversion overhead and have to deal with device variations and noise susceptibilities [19]. A programmable digital accelerator is proposed in [24] which accelerates PDEs based on multi-layer cellular nonlinear network (CeNN) engine. However, it introduces frequent external memory accesses which leads to excessive energy consumption. Mu *et al.* [32] proposed a hardware accelerator for solving 2D/3D Laplace and Poisson Equations with no external memory accesses. However, it only supports the two



(a) (b)

Figure 1: The convergence speed of FDM under different settings: (a) Gauss-Seidel with different data precision; (b) FP64 with different iteration methods.

equations with specific grid sizes. Besides customized FDM accelerators, some accelerators designed for scientific computing can also be used to solve PDEs. Feinberg *et al.* proposed an accelerator based on memristive crossbars (MemAccel) that employs BiCG-STAB [40] to solve linear system $A\vec{u} = \vec{b}$. Alrescha [1] is a recent accelerator for sparse computation that can solve PDEs using Preconditioned Conjugate Gradient (PCG) method [10]. Since the two accelerators still rely on sparse matrix-vector multiplication, they failed to exploit the computation reuse given that the sparse matrix A in FDM contains intensively repeated values. Moreover, BiCG-STAB and PCG introduce a large portion of sequential operations (23% on average in Alrescha) hindering performance improvement. Overall, a well-customized FDM accelerator is urgently needed that supports different types of PDEs with arbitrary grid sizes.

3.2.3 Opportunities for Customization. Domain-specific accelerators improve performance and energy efficiency through a combination of specialized operations, parallelism, efficient memory systems, algorithm-hardware co-design [6], which have been successfully applied to domains such as graphics, deep learning, and blockchains. For FDM, we can also design an accelerator that leverages these customization opportunities.

First, the **data representation** can be customized based on application needs, which would significantly reduce the required memory space and data accesses. Fig. 1(a) presents the convergence speed of FDM on Laplace equation with a 100 × 100 grid using different floating point formats, i.e., float16, float32 and float 64. We observed that using float32 could reach almost the same convergence speed as float64, while using float32 would take significantly more iterations to shrink the normalized residual to the same level as float64. Thus, we could use float32 to represent the data in FDM instead of float64 used in CPU implementations. Second, the computation in FDM can be distributed to multiple PEs to be **computed in parallel** to boost performance. In Jacobi method, since the u values in the previous iteration are known in the current iteration, the updating of u values can be parallelized across x and y dimensions without data dependency. Third, **data reuse** can be leveraged to

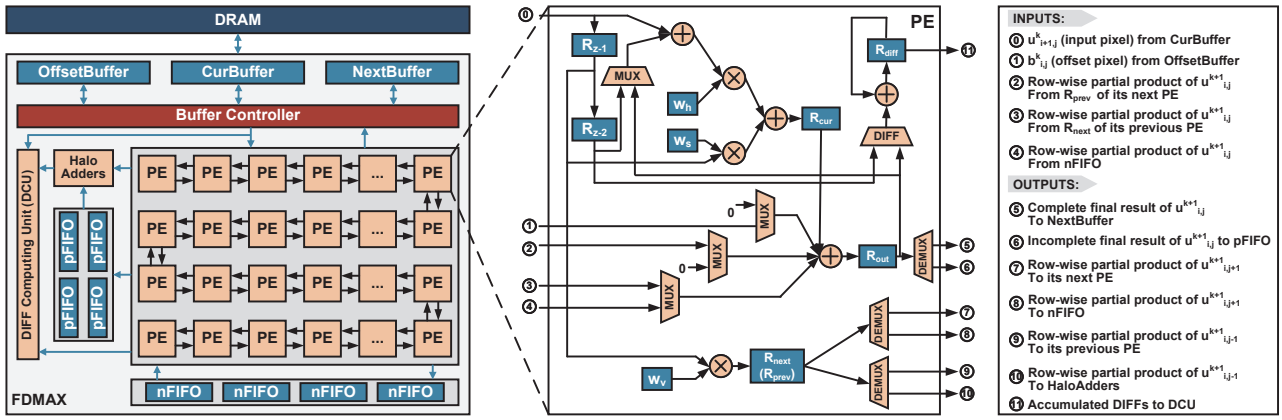


Figure 2: Full FDMAX architecture and the PE microarchitecture.

reduce data accesses by designing **dedicated datapaths**. Specifically, for Equation (6), $u^k_{i,j}$-related operations would contribute to the updating of $(u^{k+1}_{i,j}, u^{k+1}_{i-1,j}, u^{k+1}_{i+1,j}, u^{k+1}_{i,j-1}, u^{k+1}_{i,j+1})$. Hence, $u^k_{i,j}$ can be reused across these computations. Last but not least, the number of multiplications can be reduced through **dedicated arithmetic logic unit**. Specifically, CPUs and GPUs require five multiplications to update $u^k_{i,j}$ in Equation (11) since they expand the computation to matrix-vector multiplication for parallel computing. For customized accelerators, the number of multiplications can be reduced to three through tailored logic units. Such multiplication reduction could significantly improve performance and energy efficiency provided that multiplications consume more energy than additions.

In summary, there are huge demands and plenty of optimization opportunities to design an FDM accelerator for increased performance and energy efficiency.

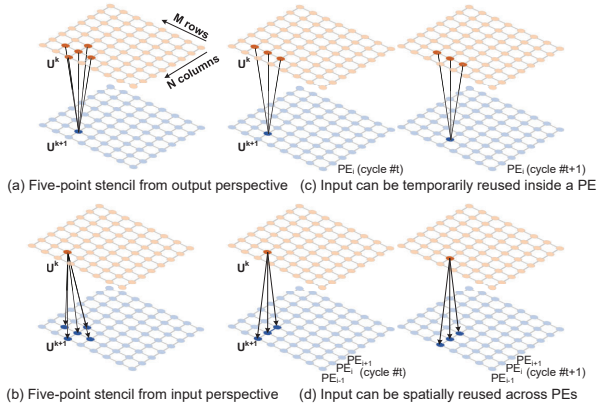


Figure 3: Forms of data reuse in FDM.

4 ARCHITECTURE DESIGN

This section presents the FDMAX accelerator to support the efficient execution of FDM. We first introduce the architecture overview, followed by the details of the PE and the PE array architecture.

4.1 Architecture Overview

Fig. 2 depicts the overall architecture of the FDMAX accelerator. FDMAX consists of the following main components: three buffers (OffsetBuffer, CurBuffer, and NextBuffer), a PE array, nFIFO, pFIFO, HaloAdders, Error Computing Unit (ECU), and a Buffer Controller. The three buffers are used to temporarily store the data required for computation (blocks of matrix B^k, U^k and U^{k+1}), thereby hiding DRAM access latency. The PE array associated with nFIFO, pFIFO, and HaloAdders is used for completing the parallel computations described in Equation (11). The ECU determines if the current solution reaches the stop condition as described in Section 2.2.5. The Buffer Controller supports the data reuse and computation in the PE array and the ECU.

To process a given FDM at the k_{th} iteration, blocks of matrix U^k and B^k are fetched from DRAM via Direct Memory Access (DMA) into CurBuffer and OffsetBuffer, respectively. The PE array fetches a portion of U^k and B^k from corresponding buffers via the buffer controller, and performs the five-point stencil computations to generate U^{k+1} block. Since some PEs may generate incomplete products at the borders of a matrix block, we use two first-in-first-out (FIFO) buffers, i.e., nFIFO and pFIFO, to temporarily store those incomplete products which will be transmitted to the PEs or HaloAdders for further computations. Since a PE receives a U_k element and generates a U_k element, PE uses the two values to compute the difference between them which contributes to the error of the current iteration. Again, the PEs generating incomplete products do not perform the difference computation. The difference computation related to the products generated by the HaloAdders is performed in the ECU. The U^{k+1} elements generated by the PE array and the HaloAdders are sent back to NextBuffer. The above steps complete the generation of the current block of U^{k+1}, then it proceeds to the next portion by repeating those steps. When all U^{k+1} elements have been updated, the ECU collected the accumulated differences from the PEs and generate the total difference of the current iteration. If the difference is less than a predefined threshold, the current U^k will be the solution, otherwise, it proceeds to the next iteration.

4.2 PE Architecture

At each cycle, each PE performs the five-point stencil computation for FDM. Although all U^k and U^{k+1} elements needed for computation are available from CurBuffer and NextBuffer, repeatedly reading them from those buffers would lead to a high energy cost

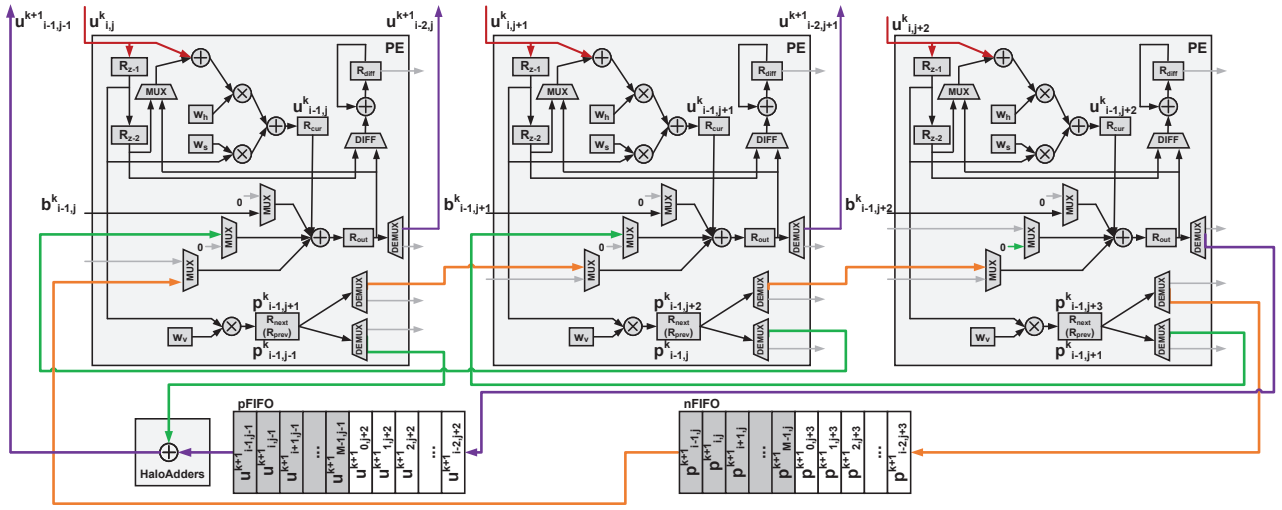


Figure 4: PE connection with neighbor PEs. Red lines: input grid values. Green/Orange lines: fetching partial product from next/previous PEs.

and require a high bandwidth. Enabling data reuse could substantially reduce the traffic between PEs and on-chip buffers. Fig. 3(a) and 3(b) present the two data reuse forms in FDM computation. One is output reuse in which an output element needs to collect the multiplication results from five input elements in U^k. The other is input reuse where one input element contributes to five output elements.

It is hard to simultaneously maximize input and output reuse because the multiplication results generated using the same input element are not reducible. To address this problem, we split five-point stencil computation into column-wise and row-wise computations, as shown in Fig. 3(c) and 3(d). The column-wise computations are assigned to the same PE so that the PEs can reuse input temporarily as shown in Fig. 3(c). Meanwhile, the output is partially reused since the PE collects three multiplication results in one cycle. Neighbor columns are assigned to neighbor PEs as shown in Fig. 3(d) so that the input can be spatially reused since it can contribute to the row-wise computations by enabling inter-PE propagation.

4.2.1 Five-point Stencil Computation. Fig. 2 presents the PE architecture designed to support the data reuse patterns described above. The column-wise computation is like a 1D convolution with a kernel size of 3. To exploit the sliding window feature of the column-wise computations, we added two registers (R_{z-1} and R_{z-2}) to the PE to store the past two input elements in a sliding window. Registers W_h, W_s and W_v store the corresponding weight values and remain unchanged through the entire FDM process. Input port 0 receives one input element in each cycle, which is used to perform the column-wise computations together with R_{z-1}, R_{z-2}, W_h and W_s. The PE also reads the offset B^k element (Input port 1) from OffsetBuffer and added it to the column-wise product (R_{cur}). R_{z-1} and R_{z-2} work in a systolic fashion to proceed to the next sliding window thereby eliminating the need to repeatedly read input elements from CurBuffer. For the row-wise computation, the PE collects the multiplication results generated in neighbor PEs via Input ports 2 and 3, and then adds with the column-wise product to generate the final product (R_{out}). In other words, the PE also needs to generate row-wise products for its neighbor PEs. Specifically, the PE multiplies R_{z-1} with W_v to generate a row-wise partial product, and sends it to neighbor PEs via Output port 7 and 9. The final product is sent back to NextBuffer via Output port 5.

We split the PE computations into a two-stage pipeline to shorten the critical path. The first stage is the operations generating column-wise and row-wise products (R_{next} and $R_{next}(R_{prev})$). The second is the final product assembling and the DIFF computation.

4.2.2 PE Connection. Since border PEs only have one neighbor, they cannot send and receive row-wise partial products to and from two neighbor PEs. Specifically, the first PE cannot receive and send row-wise partial product from and to its previous PE, while the last PE cannot receive and send row-wise partial product from and to its next PE. Fig. 4 presents an example of FDMAX with three PEs. We denote the leftmost PE as the first PE and the rightmost PE as the last one. The orange lines show the propagation of row-wise partial products to the next PEs, and the green lines show the propagation of row-wise partial products to previous PEs. The middle PE has two neighbor PEs so that it receives two row-wise partial products and generates a complete final result ($u^{k+1}_{i-1,j+1}$). Meanwhile, the middle PE can also send its generated row-wise partial product to its two neighbor PEs. For the last PE, it cannot send and receive the row-wise partial products to and from its next PE. As a result, R_{out} stores an incomplete final product and it needs extra buffers to store R_{next}. Therefore, we added the nFIFO and pFIFO buffers to temporarily store the incomplete final product and row-wise partial product, respectively. For the first PE, it cannot send and receive the row-wise partial products to and from its next PE. Its required row-wise partial product can be read from nFIFO as they are stored in nFIFO in the previous column batch. Its generated row-wise partial product has to be added to the incomplete final product stored in pFIFO to generate the complete final product, which is performed in the HaloAdders.

To differentiate the role of middle PEs and border PEs, we added multiplexers and demultiplexers in the PE to control the source and destination of the input and output ports. Specifically, the last

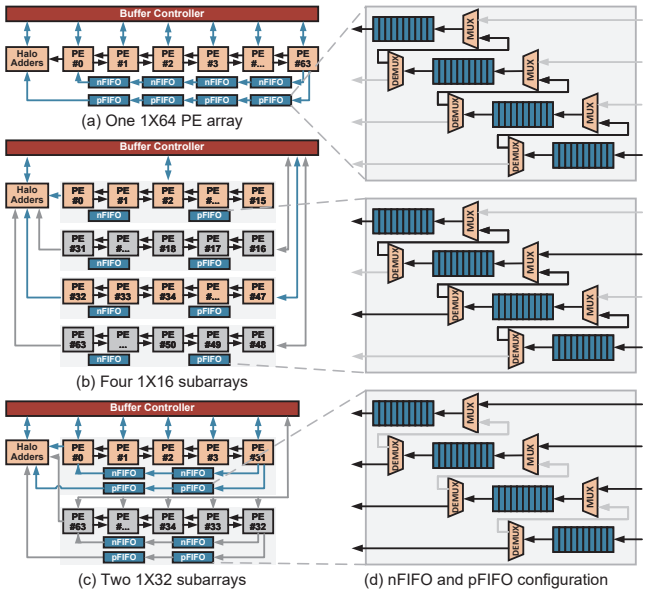


Figure 5: The elastic PE array architecture and nFIFO/pFIFO structure.

PE forwards the incomplete final product to pFIFO via Output port 6, and forwards row-wise partial product to nFIFO via Output port 8. Since the first PE does not have a previous PE, it reads the row-wise partial product generated in the previous column batch from nFIFO. Meanwhile, the first PE forwards the row-wise partial product via Output port 10 to HaloAdders instead of Output port 9. The HaloAdders add the incomplete final products from pFIFO and the row-wise partial products to resolve the halo problem between column batches. The details will be further explained in Section 5.

4.2.3 Update method. As analyzed in Section 2.2, using the latest u values available to update subsequent values can speed up convergence, but it sacrifices parallelism. In our PE architecture, the latest value generated from R_{out} can be immediately used to replace the value stored in R_{z-2}. Hence, we added a multiplexer to provide the PE flexibility to use R_{z-2} or R_{out} for subsequent computations. Therefore, PE can support both Jacobi and hybrid methods.

4.2.4 Stop condition. As described in Section 2.2.5, the iteration stops when the differences between U^{k+1} and U^k reach a stop condition. Prior work [32] explicitly specifies the number of iterations and relies on a host CPU to determine if the solution converges, which increases the communication cost between the chip and CPU. In FDMAX, we added a DIFF logic inside each PE to accumulate the square of the difference between U^{k+1} and U^k. As shown in Fig. 2, the DIFF logic reads the U^k element from R_{z-2} and R_{out} and generates the square of their difference which is then accumulated to R_{diff}. The details of the DIFF logic is eliminated for simplicity.

4.3 Elastic PE Array Architecture

FDMAX employs multiple PEs operating in parallel and communicating with neighbors to boost the performance of processing FDM. Given that the on-chip buffers have limited storage space, the FDM has to be partitioned and mapped to the accelerator. Specifically, the size of on-chip buffers is usually 10 to 100 KB, while an FDM with a 1000 ×1000 grid requires tens of MB. Moreover, computational characteristics of different PDE-solving problems vary in data reuse and coarse-grained parallelism because they have different grid and step sizes. Specifically, given that each PE accommodates a column of the matrix as described in Section 4.2 and Fig. 3, it would lead to resource under-utilization when the number of columns is less than the number of PEs, e.g., a tall-and-thin matrix. In such cases, we can split the rows into multiple groups and each group is allocated to a PE so that multiple PEs accommodate a column in parallel.

It requires the accelerator to support flexible dataflow to adapt to different PDEs. Fortunately, our PE array and buffer structure can easily support this feature by dynamically decomposing the PE array into several PE subarrays. Fig. 5 presents three possible configurations for the decomposition of a 4 ×16 PE array, where a 1×16 subarray is used as the granularity for reconfiguration. Fig. 5(a) shows a reconfiguration example where all the PEs combine to a 1 ×64 array, which is good at processing short and fat grids. In Fig. 5(b), the PE array is reconfigured to form four 1 ×16 subarrays for processing tall and thin grids. Alternatively, the PE array can be reconfigured to other shapes as shown in Fig. 5(c), where the PE array is reconfigured to two subarrays of size 1 ×32. Since there is an order between the PEs as described in Section 4.2.2, it should be noted that for the PEs in even-indexed rows such as the second and fourth rows, its right-hand PE serves as its previous PE while its left-hand PE as its next PE.

Each PE can act as the first PE, middle PE, or last PE by configuring the MUXes and DEMUXes as described in Section 4.2. Therefore, the reconfiguration of the PE array can be easily achieved through the configuration of the PEs. Besides the PE array, the nFIFO and pFIFO structures also require reconfiguration to support the subarrays. Fig. 5(d) illustrates the reconfiguration of the FIFOs for the three scenarios of the PE array configuration. By adding MUXes and DEMUXes to the FIFO structures, nFIFO and pFIFO can be reconfigured to multiple sub-FIFOs to serve the PE subarrays.

5 FDM MAPPING

In this section, we use Laplace equation as an example to show how to map FDM to FDMAX. We present in Fig. 6 to illustrate how FDMAX computes U^{k+1} elements in parallel. Without losing any generality, we consider a small design with a 1 ×3 PE array (PE_0, PE_1 and PE_2), where PE_0 is the first PE and PE_2 is the last one. The grid size is 100 ×100. The mapping adapts to both Jacobi and hybrid methods. We omit the irrelevant logic to Laplace equations for clarity. Specifically, the terms $c_{i,j}$ and w_s in Equation (11) remain zero values in Laplace equation so they are not depicted in the figure. The DIFF module is also omitted as it does not affect the understanding of the mapping. The weight values (w_v and w_h) are prestored in the respective registers and keep unchanged for all iterations, while the other registers are initialized to be zero. The computation is pipelined to reduce critical path. The PE array works as follows:

Cycle #0: All three PEs respectively read their input elements ($u^0_{0,0}$, $u^0_{0,1}$ and $u^0_{0,2}$) from CurBuffer. This input element adds the R_{z-2} value (initialized as zero) and the addition result multiplies w_v to generate the column-wise product, which is then stored in

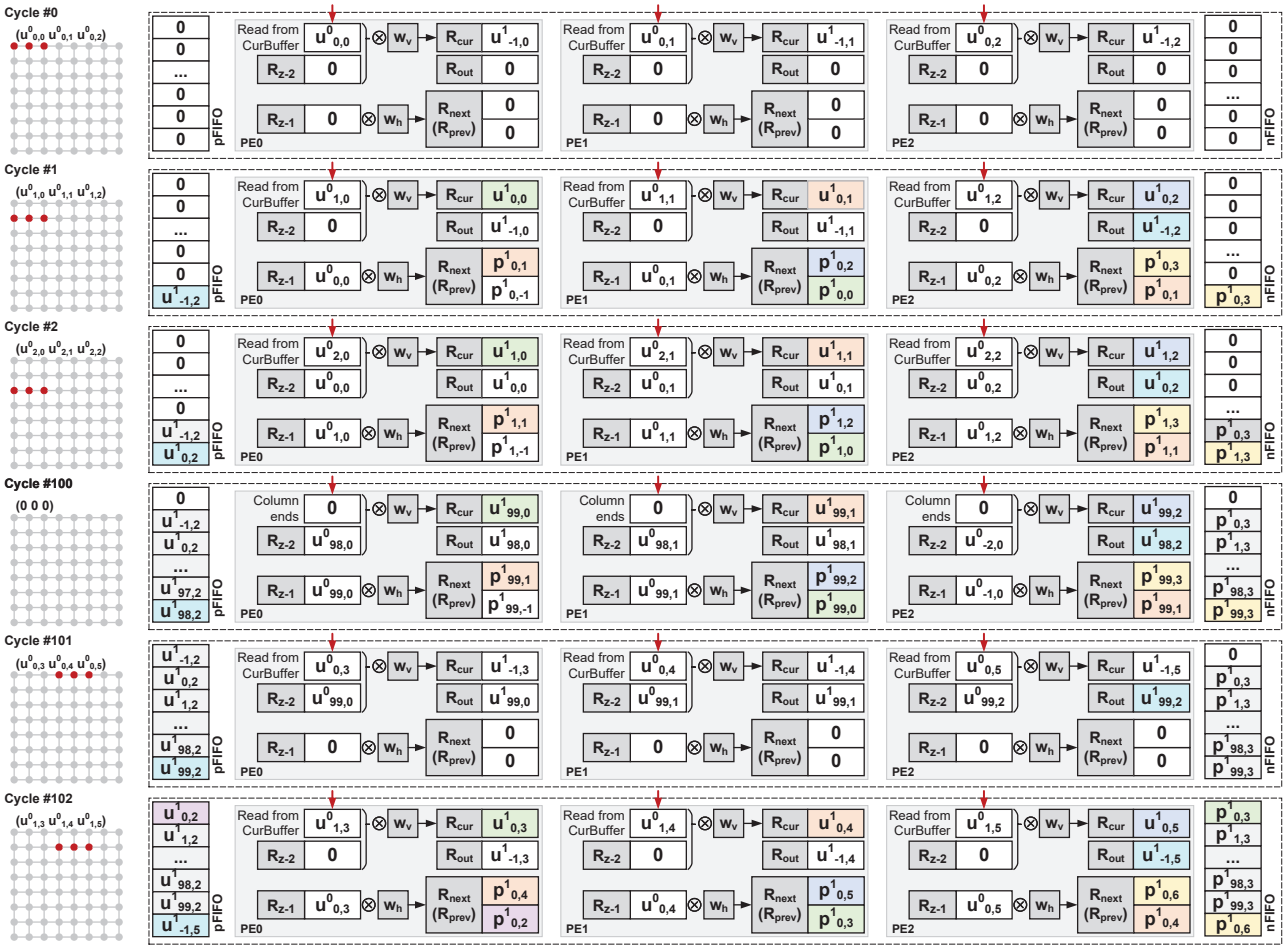


Figure 6: Mapping Laplace equation to FDMAX with a 1×3 PE array.

R_{cur}. Meanwhile, the input element will be stored in R_{z-1} while the original R_{z-1} value is forwarded to update R_{z-2}. Note that since this cycle is at the warm-up period, we use $u^1_{-1,0}$, $u^1_{-1,1}$ and $u^1_{-1,2}$ to represent the result stored in R_{cur}, which are actually meaningless and useless in updating U^1. Nevertheless, these values have to be used for the correctness of the computation.

Cycle #1: We depict the operations stage by stage in the pipeline. For the final product assembling, the middle PE (PE_1) collects its R_{cur}, the R_{prev} from PE_2, the R_{next} from PE_0 to generate the final product and stores it in R_{out} ($u^1_{-1,1}$). The first PE (PE_0) reads from nFIFO instead of R_{prev} to generate the final product ($u^1_{-1,0}$). The last PE (PE_2) generates an incomplete final product ($u^1_{-1,2}$) that is sent to pFIFO. For the column- and row-wise computation stage, the three PEs read input elements ($u^0_{1,0}$, $u^0_{1,1}$ and $u^0_{1,2}$), and perform computations similar to those in Cycle #0. R_{z-1} and R_{z-2} are also updated in a systolic fashion as in Cycle #0. In PE_2, the generated row-wise product ($p^1_{0,3}$) is forwarded to nFIFO, which will be used when the PEs switch to the next column batch.

Cycle #2 to #99: The PEs repeatedly move to the next rows in each cycle and perform similar operations in Cycle #1.

Cycle #100: The PEs have already swept all the rows through the previous cycles. Before switching to the next column batch, we insert a NULL cycle at here where PEs read zeros as input elements to flush out the final product of current columns.

Cycle #101: The PEs switch to next the three columns and read their input elements ($u^0_{0,3}$, $u^0_{0,4}$ and $u^0_{0,5}$) from CurBuffer. The final product assembling stage generates the U^{k+1} values of the last row ($u^1_{99,0}$, $u^1_{99,1}$ and $u^1_{99,2}$).

Cycle #102: The PEs perform similar operations as those in Cycle #1. The difference lies in the pFIFO and nFIFO. Beginning at this cycle, the R_{prev} in PE_0 is sent to HaloAdders to be added with the incomplete final product stored in pFIFO. Specifically, $p^1_{0,2}$ in PE_0 is added with $u^1_{0,2}$ in pFIFO to generate the complete final product of $u^1_{0,2}$. Also, PE_0 reads the row-wise product of the previous row from nFIFO to assemble the final product. Specifically, $p^1_{0,3}$ in nFIFO is sent to PE_0 to contribute to the final product $u^1_{0,3}$.

The following cycles are not depicted for sake of brevity. In this example, input U^k matrix is only fetched once to the PE array, which significantly reduces the number of reads from CurBuffer and the internal bandwidth between buffers and the PE array. The nFIFO and pFIFO structures contribute to the main overhead of the implementation. However, this overhead could be significantly amortized as there are usually tens to hundreds of PEs in practice.

6 EXPERIMENTAL SETUP

6.1 FDMAX Configuration

While the FDMAX accelerator can be scaled across a number of dimensions, we investigate an FDMAX design with the following key parameters. The design is equipped with an 8×8 PE array, and eight nFIFOs and eight pFIFOs accordingly. Each nFIFO or pFIFO has 64 entries. We set the clock frequency at 200 MHz according to the synthesis result. We use High Bandwidth Memory (HBM) as off-chip DRAM that delivers bandwidth up to 128 GB/s. The CurBuffer, OffsetBuffer, and NextBuffer are banked to support the concurrent data accesses of the PEs as well as match the DRAM bandwidth. Specifically, considering an ideal case that the DRAM bandwidth is fully utilized, the number of data elements that can be transmitted between DRAM and on-chip buffers in one FDMAX's clock cycle is 160 ((128GB/s)/(200MHz*4Bytes)). Meanwhile, the PE reads one or two data elements and write one from and to the buffers in each cycle resulting in traffic of $(1+1)*8*8 \sim (2+1)*8*8$ between the PE array and the buffers, i.e., $128 \sim 192$ data elements per cycle. Residing between DRAM and the PE array, the on-chip buffers should provide comparable bandwidth to keep pace with them. However, heavily banked buffers would lead to a large area overhead. We observed that each buffer with 32 banks makes an optimal balance between performance and overhead. We set the depth of the buffers at 32, so the capacity of each buffer is 4 KB.

6.2 Methodology

The performance and energy of FDMAX are measured using the following tools.

Architecture Simulator. We implemented a cycle-accurate simulator to model the hardware behavior. The simulator models the microarchitectural behavior of each hardware module, and counts the exact number of execution cycles and data accesses including DRAM accesses, on-chip buffer accesses, and Register File accesses. According to the mapping mechanism of FDM described in section 5, the simulator models each module down to the microarchitectural level, i.e., the registers, the multiplexers/demultiplexers, the adders/multipliers, etc. Hence, we can accurately model the number of cycles needed to map FDMs onto the accelerator. We also obtained the number of read/write of each SRAM and FIFO from the simulator and used CACTI 6.5 to estimate the energy and area of SRAMs and FIFOs.

ASIC Synthesis. To measure the area and power consumption of the accelerator, we implemented and synthesized each module in Verilog. We use the Synopsys Design Compiler with Synopsys Generic SAED 32nm library for the synthesis, and estimate the power using Synopsys PrimeTime PX. We use CACTI 6.5 to estimate the energy cost of DRAM accesses.

6.3 Benchmark PDEs

We use the PDEs listed in Table 1 as the benchmarks to evaluate FDMAX. These equations are specified with the Dirichlet Boundary Conditions [13]. We set all grid values at zero as the initial conditions. We investigate grid sizes ranging from 100×100 to $10K \times 10K$.

6.4 Baseline Platforms

We compare FDMAX with CPUs and GPUs in terms of performance and energy consumption. We implement FDM in python on a Linux server equipped with Intel Xeon Gold 6226R CPU@2.90 GHz with 22 MB LLC. We also implement FDM in CUDA C/C++ based on the open-source code provided by Nvidia [16] and evaluate on an enthusiast-class enthusiast-class graphics card, NVIDIA GeForce RTX 3090, which is equipped with 24 GB GDDR6X memory and 936.2 GB/s. We also implement the checkerboard update method on GPU following the methodology proposed in [11]. Since the objective of Heat and Wave equations is to obtain the grid values at each time step, the convergence speed is not applicable to the two equations. Hence, we only evaluate the Jacobi method on the two equations. We set the same stop conditions for Laplace and Poisson equations, and the same number of iterations for Wave and Heat equations to make a fair comparison. We use the letters $\{J, G, H, C\}$ attached to the platform to denote different update methods. For example, CPU-J denotes using Jacobi method for FDM on CPU. There are two ways to implement the Jacobi method of FDM on CPU, the five-point stencil style as shown in Equation (11) and matrix multiplication form described in Section 2.2. We observed that for small grid size such as 100×100, the matrix multiplication form is faster since it can leverage the SIMD cores libraries for matrix processing to boost performance. However, for large grid size such as 1000×1000, the matrix multiplication form introduces a stencil matrix A of size $1M \times 1M$, which is not practical to implement on CPU. Therefore, we use the five-point stencil form for CPU-J in the evaluation. To evaluate the energy consumption of CPU, we use the Average CPU Power (ACP) [22] multiplied by the processing time to generate CPU energy. The energy consumption of GPU is estimated using the NVIDIA Power Capture Analysis Tool (PCAT).

Besides CPU and GPU, we also compare FDMAX with the state-of-the-art hardware accelerators for solving PDEs, including BitSerial [33], MemAccel [12] and Alrescha [1] as listed in Table 2. Since these accelerators are not open-sourced, we model their behaviors based on the published papers. These accelerators are equipped with the same computation and memory bandwidth budget (128 GB/s), and also use single precision floating point numbers for a fair comparison. As MemAccel and Alrescha use different PDE-solving methods, we estimate their execution time by multiplying the required number of iterations and the latency for each iteration. The required number of iterations of BiCG-STAB (used by MemAccel) and PCG (used by Alrescha) are derived from the CPU implementation. We also collect the numbers of computational and memory operations of the baseline accelerators and combine them with the energy values reported in [20] to estimate the energy consumption. Our execution time and energy numbers are validated based on the reported numbers in the original paper for their configurations to ensure that the reproduced numbers are never worse than their reported numbers.

7 EXPERIMENTAL RESULTS

7.1 Layout Characteristics

Table 3 presents the layout characteristics of FDMAX. The total area and power of FDMAX are 0.99 mm^2 and 1711.27 mW, respectively. The on-chip storage structures consume the most of power and

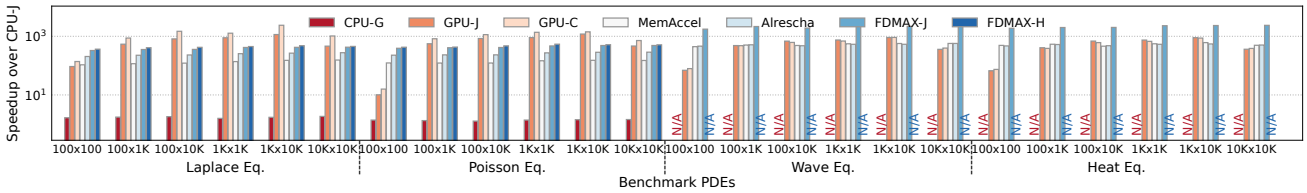


Figure 7: Speedup up of the PDE solvers (CPU-G, GPU-J, GPU-C, MemAccel, Alrescha, and FDMAX) over CPU-J.

Table 3: Layout characteristics of FDMAX

Component	Size	Area (mm^2)	Power (mW)
PE Array	8 ×8 PEs	0.047 (4.79%)	293.04 (17.12%)
Buffer Controller	-	0.020 (2.01%)	18.72 (1.09%)
nFIFO	8 ×64 entries	0.10 (10.06%)	142.90 (8.35%)
pFIFO	8 ×64 entries	0.10 (10.06%)	142.20 (8.31%)
CurBuffer	4 KB	0.24 (24.36%)	373.61 (21.83%)
OffsetBuffer	4 KB	0.24 (24.36%)	369.25 (21.58%)
NextBuffer	4 KB	0.24 (24.36%)	371.55 (21.71%)
Total	-	0.99 (100%)	1711.27 (100%)

area. Specifically, CurBuffer, OffsetBuffer and NextBuffer contribute to 73.08% and 65.12% of the total area and power respectively because they are heavily banked to support concurrent data accesses. Besides, a significant fraction of the area is contributed by nFIFO and pFIFO. The PE array consumes 17.12% of the power with 4.79% of the area.

7.2 Speedup

Fig. 7 shows the speedup of FDMAX and the baselines. For Jacobi method, FDMAX-J achieves 1260×, 1189×, 5.8×, 4.9×, 3.6×, and 2.9× speedup on average over CPU-J, CPU-G, GPU-J, GPU-C, MemAccel, and Alrescha respectively. The performance improvement stems from the customized PE and datapath designs that leverage parallel processing with reduced data access latency. First, the PE array employs multiple PEs operating in parallel to speed up the computations. Second, FDMAX leverages computation reuse to reduce the number of multiplications since multiplication is much more complicated than addition. Third, the PEs operate in a systolic fashion which significantly reduces the traffic between PEs and on-chip buffers, thereby reducing the data access latency of the PEs. Finally, the nFIFO and pFIFO structures temporarily store the intermediate data that are reused in future computations to further reduce the traffic between PEs and the buffers (CurBuffer and NextBuffer). Although MemAccel and Alrescha also leverage customized structures for parallelism, they primarily focus on solving systems of linear equations other than FDM, hence they failed to exploit the computation reuse in FDM which hinders performance improvement. Moreover, the algorithms they used to solve the equations would introduce sequential operations (up to 30% for Alrescha) and complex computational operations (SpMV and SymGS in Alrescha) which harms performance in two aspects: 1) the sequential operations and complex computational operations require extra hardware that occupies computing resources. Since the baseline accelerators are provisioned with the same hardware resource budget, the hardware resource budget for each operation type is undercut for a fair comparison which leads to performance degradation. 2) the sequential operations limit their maximum expected improvements according to Amdahl's law. By contrast, FDMAX can almost process all the operations in FDM in parallel. Although BiCG-STAB and PCG achieve faster convergence than Jacobi and Gauss-seidel method [8], the results show that it cannot cover the overhead when considering hardware implementation.

In terms of specific PDEs, FDMAX achieves better speedup on Wave and Heat equations than Laplace and Poisson Equations. The reason is that FDMAX runs more iterations on Laplace and Poisson equations to reach the stop condition than CPU and GPU because FDMAX uses 32-bit floating point numbers. We observed FDMAX-J and FDMAX-H run about 80% and 60% more iterations compared to CPU-J. GPU achieves higher performance than FDMAX in some cases for Laplace and Poisson equations, which mainly attributes to its much higher memory bandwidth at 936.2 GB/s compared to FDMAX (128 GB/s). Nevertheless, FDMAX still outperforms GPU in most cases and by 5.8× on average. In terms of update methods, FDMAX-H outperforms FDMAX-J (1.05× faster on average) because of hybrid method reaches a faster convergence than Jacobi method does. Meanwhile, GPU-C performs 1.2× better than GPU-J on average.

7.3 Energy Consumption

We compare the energy consumption in Fig. 8. The energy consumption evaluated includes those consumed by the off-chip DRAM accesses. FDMAX-H consumes on average only 0.06%, 0.09%, 11.7%, 17.3%, 55.7%, and 65.9% energy compared to CPU-J, CPU-G, GPU-J, GPU-G, MemAccel and Alrescha, respectively. The energy reduction of FDMAX stems from three aspects. First, FDMAX has much fewer off-chip data accesses because of the datapath and data reuse optimizations. Second, the systolic-style PE array substantially reduces the number of on-chip buffer accesses since on-chip SRAM access also consumes a large portion of energy consumption. Third, the PE reduces the number of multiplications which saves energy since multiplication consumes more energy than addition.

7.4 Scalability and Sensitivity Analysis

FDMAX can be scaled across many dimensions, among which the size of PE array is the most important one. Therefore, we conduct a scalability study by investigating FDMAX equipped with a PE array of size $S \times S$, S=4,5,...12. In Fig. 9, we show how FDMAX versions with different PE array sizes perform under the DRAM bandwidth sweeping from 16 GB/s to 256 GB/s (Fig. 9(a)), and number of buffer banks sweeping from 8 to 64 (Fig. 9(b)). The benchmark is the Laplace equation with $10K \times 10K$ grid using Jacobi method. The nFIFO and pFIFO scale with the PE array size. It should be noted

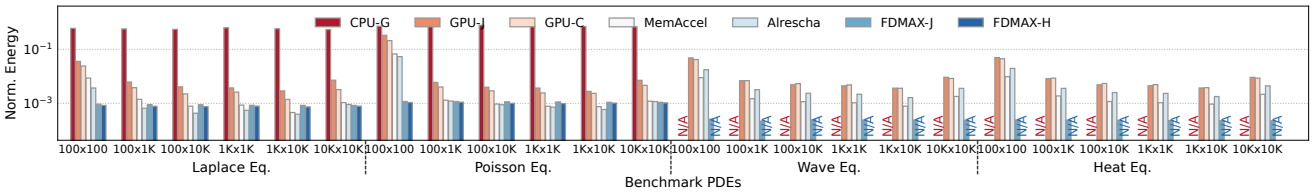
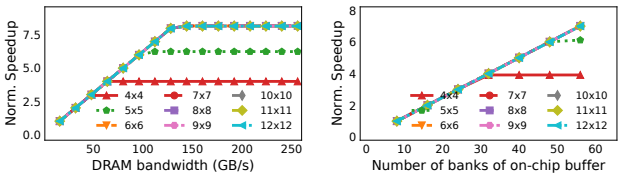


Figure 8: Normalized energy consumption of the PDE solvers (CPU-G, GPU-J, GPU-C, MemAccel, Alrescha, and FDMAX) over CPU-J.



(a) (b)

Figure 9: Scalability study of FDMAX to the scale of PE array. (a) number of on-chip buffer bank keeps constant at 64; (b) DRAM bandwidth keeps constant at 256GB/s.

that FDMAX can be easily replicated to form larger PE arrays with negligible overhead on interconnection.

It is obvious that higher performance can be achieved with larger PE arrays. When the PE array size is smaller than 7×7 with a bandwidth of 256 GB/s, the performance of FDMAX grows almost linearly with the increasing of PE array size, which proves the good scalability of FDMAX. When the PE array is larger than 8×8, the performance gain is marginal with larger PE array because DRAM and SRAM bandwidth become the bottleneck. It is clearly that the performance of all FDMAX versions improves with the increasing of DRAM bandwidth and number of buffer banks, which again emphasizes the importance of optimization of data accesses.

7.5 Comparison with Prior FDM Accelerators

BitSerial [33] is a recent FDM accelerator for solving 2D/3D PDEs. However, it is hardly possible to make a fair quantitative comparison between FDMAX and BitSerial. First, Bitserial assumes an identical step size for each dimension thus replacing multiplications with additions at the cost of losing generality. Second, BitSerial only supports specific grid sizes which further limits its application. Nevertheless, we think comparing the reported statistics listed in Table 2 to make a qualitative evaluation is still informative. Again, we would like to highlight the merit of FDMAX that it efficiently supports different PDE types with arbitrary grid size, which is the main advantage over prior approaches.

Since Checkerboard method achieves faster convergence than Hybrid method does, we would like to explain why FDMAX chooses Hybrid method other than Checkerboard method. Compared to Jacobi or Hybrid method, the faster convergence of Checkerboard is at the cost of parallelism as demonstrated in [33]. Specifically, only half of the PEs can update the grid values at each cycle in Checkerboard method, while Jacobi/Hybrid method can update all the grid values in a single cycle. We observed that to reach a given error tolerance, Hybrid method needs no more than 1.4× as many iterations as Checkerboard method does. Therefore, using Hybrid method will achieve better overall efficiency in FDMAX.

8 RELATED WORK

Plenty of work target PDE solving based on CPU- and GPU-accelerated computation [2, 3, 9, 14, 15, 21, 26, 27, 29, 30, 35]. Specifically, GdfidL [3] uses linked lists instead of three 3D arrays for finite difference programs to reduce memory and CPU usage. Micikevicius [30] proposed a GPU parallelization of the 3D difference computation using CUDA. However, it is only targeting FDM with the same step size for the three dimensions. These implementations utilizes the thousands of threads in GPUs, however, they are power-hungry and energy-inefficient, making them hardly applicable for scenarios with limited power and energy budgets.

While domain-specific accelerators have become one of the most important ways to achieve increased performance and energy efficiency in post-Moore's Law era, researchers have proposed customized accelerators to solve PDEs except the ones introduced in Section 3.2. Durbano *et al.* [7] presented FPGA-based accelerator for 3D FDTD algorithms with optimizations on the memory hierarchy, and the computational datapath. Ishgaki *et al.* [23] presented an FPGA implementation of 3D numerical simulations on a 2D SIMD array processor. Waidyasooriya *et al.* [41] designed FPGA-based pipelined architecture for FDTD acceleration using OpenCL. Although these approaches achieved substantial speedup over general-purpose processors, they usually target on a specific equation and cannot support a wide range PDEs.

9 CONCLUSION

This paper presents the FDMAX architecture to efficiently accelerate FDM for solving PDEs. FDMAX exploits a reconfigurable PE design to efficiently support the computation patterns in a broad range of PDEs. The PEs are easily replicated to form large PE arrays for increased performance with negligible interconnection overhead. The PE array operates in a systolic fashion to significantly reduce the data accesses. In addition, the PE array dynamically breaks into several subarrays to adapt to different grid sizes. Averaged across four PDEs with different grid sizes, FDMAX demonstrates significant performance and energy efficiency improvements compared to CPUs, GPUs and the state-of-the-art PDE accelerators. We believe that our work is a good practice for domain-specific accelerators in the field of solving PDEs, and would potentially open up new opportunities for fields like real-time simulation.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under Grant No. 62202020, and in part by the National Science Foundation under Grant CCF-2245950.

REFERENCES

[1] B. Asgari, R. Hadidi, T. Krishna, H. Kim, and S. Yalamanchili. 2020. ALRESCHA: A Lightweight Reconfigurable Sparse-Computation Accelerator. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 249–260. https://doi.org/10.1109/HPCA47549.2020.00029

[2] Ana Balevic, Lars Rockstroh, Andreas Tausendfreund, Stefan Patzelt, Gert Goch, and Sven Simon. 2008. Accelerating simulations of light scattering based on finite-difference time-domain method with general purpose GPUs. In *2008 11th IEEE International Conference on Computational Science and Engineering*. IEEE, 327–334.

[3] Warner Bruns. 1997. GdfidL: A finite difference program with reduced memory and CPU usage. In *Proceedings of the 1997 Particle Accelerator Conference (Cat. No. 97CH36167)*, Vol. 2. IEEE, 2651–2653.

[4] T. Chen, J. Botimer, T. Chou, and Z. Zhang. 2020. A 1.87-mm2 56.9-GOPS Accelerator for Solving Partial Differential Equations. *IEEE Journal of Solid-State Circuits* 55, 6 (2020), 1709–1718. https://doi.org/10.1109/JSSC.2019.2963591

[5] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. 2016. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the eleventh european conference on computer systems*. 1–16.

[6] William J Dally, Yatish Turakhia, and Song Han. 2020. Domain-specific hardware accelerators. *Commun. ACM* 63, 7 (2020), 48–57.

[7] James P Durbano and Fernando E Ortiz. 2004. FPGA-based acceleration of the 3D finite-difference time-domain method. In *12th Annual IEEE symposium on field-programmable custom computing machines*. IEEE, 156–163.

[8] Adam Dziekonski, Adam Lamecki, and Michal Mrozowski. 2010. Jacobi and Gauss-Seidel preconditioned complex conjugate gradient method with GPU acceleration for finite element method. In *The 40th European Microwave Conference*. IEEE, 1305–1308.

[9] Adam Dziekonski, Piotr Sypek, Adam Lamecki, and Michal Mrozowski. 2012. Finite element matrix generation on a GPU. *Progress In Electromagnetics Research* 128 (2012), 249–265.

[10] Stanley C Eisenstat. 1981. Efficient implementation of a class of preconditioned conjugate gradient methods. *SIAM J. Sci. Statist. Comput.* 2, 1 (1981), 1–4.

[11] M. ElMaghrbay, R. Ammar, and S. Rajasekaran. 2013. Fast GPU algorithms for implementing the red-black Gauss-Seidel method for Solving Partial Differential Equations. In *2013 IEEE Symposium on Computers and Communications (ISCC)*. 000269–000274. https://doi.org/10.1109/ISCC.2013.6754958

[12] B. Feinberg, U. K. R. Vengalam, N. Whitehair, S. Wang, and E. Ipek. 2018. Enabling Scientific Computing on Memristive Accelerators. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 367–382. https://doi.org/10.1109/ISCA.2018.00039

[13] Gerald B Folland. 2020. *Introduction to partial differential equations*. Princeton university press.

[14] Zhisong Fu, T James Lewis, Robert M Kirby, and Ross T Whitaker. 2014. Architecting the finite element method pipeline for the GPU. *Journal of computational and applied mathematics* 257 (2014), 195–211.

[15] Mike Giles, Endre László, István Reguly, Jeremy Appleyard, and Julien Demouth. 2014. GPU implementation of finite difference solvers. In *2014 Seventh Workshop on High Performance Computational Finance*. IEEE, 1–8.

[16] Github. 2016. NVIDIA-developer-blog. https://github.com/NVIDIA-developer-blog/code-samples/blob/master/series/cuda-cpp/finite-difference/finite-difference.cu

[17] Dimitris A Goussis and Ulrich Maas. 2011. Model reduction for combustion chemistry. *Turbulent combustion modeling: advances, new trends and perspectives* (2011), 193–220.

[18] Boyce E Griffith and Xiaoyu Luo. 2017. Hybrid finite difference/finite element immersed boundary method. *International journal for numerical methods in biomedical engineering* 33, 12 (2017), e2888.

[19] Ning Guo, Yipeng Huang, Tao Mai, Sharvil Patil, Chi Cao, Mingoo Seok, Simha Sethumadhavan, and Yannis Tsividis. 2016. Energy-efficient hybrid analog/digital approximate computation in continuous time. *IEEE Journal of Solid-State Circuits* 51, 7 (2016), 1514–1524.

[20] Mark Horowitz. 2012. Energy table for 45nm process. Available:https://sites.google.com/site/seecproject

[21] Peter Huthwaite. 2014. Accelerated finite element elastodynamic simulations using the GPU. *J. Comput. Phys.* 257 (2014), 687–707.

[22] Intel. 2011. Measuring Processor Power: TDP vs. ACP. Available:https://www.intel.com/content/dam/doc/white-paper/resources-xeon-measuring-processor-power-paper.pdf

[23] Yutaro Ishigaki, Yoichi Tomioka, Tsugumichi Shibata, and Hitoshi Kitazawa. 2015. An FPGA implementation of 3D numerical simulations on a 2D SIMD array processor. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 938–941.

[24] Jaeha Kung, Yun Long, Duckhwan Kim, and Saibal Mukhopadhyay. 2017. A Programmable Hardware Accelerator for Simulating Dynamical Systems. *SIGARCH Comput. Archit. News* 45, 2 (2017), 403–415. https://doi.org/10.1145/3140659.3080252

[25] Harvard Lomax, Thomas H Pulliam, David W Zingg, Thomas H Pulliam, and David W Zingg. 2001. *Fundamentals of computational fluid dynamics*. Vol. 246. Springer.

[26] Graham R. Markall, David A. Ham, and Paul H. J. Kelly. 2010. Towards generating optimised finite element solvers for GPUs from high-level specifications. *Procedia Computer Science* 1, 1 (2010), 1815–1823. https://doi.org/10.1016/j.procs.2010.04.203

[27] Jesús Martínez-Frutos and David Herrero-Pérez. 2015. Efficient matrix-free GPU implementation of fixed grid finite element analysis. *Finite Elements in Analysis and Design* 104 (2015), 61–71.

[28] Dagmar Medková. 2018. The Laplace Equation. *Boundary value problems on bounded and unbounded Lipschitz domains. Springer, Cham* (2018).

[29] David Michéa and Dimitri Komatitsch. 2010. Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards. *Geophysical Journal International* 182, 1 (2010), 389–402.

[30] Paulius Micikevicius. 2009. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd workshop on general purpose processing on graphics processing units*. 79–84.

[31] Juan Pedro Milaszewicz. 1987. Improving jacobi and gauss-seidel iterations. *Linear Algebra and Its Applications* 93 (1987), 161–170.

[32] J. Mu and B. Kim. 2021. 29.2 A 21×21 Dynamic-Precision Bit-Serial Computing Graph Accelerator for Solving Partial Differential Equations Using Finite Difference Method. In *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, Vol. 64. 406–408. https://doi.org/10.1109/ISSCC42613.2021.9366053

[33] Junjie Mu, Chengshuo Yu, Tony Tae-Hyoung Kim, and Bongjin Kim. 2022. A Scalable Bit-Serial Computing Hardware Accelerator for Solving 2D/3D Partial Differential Equations Using Finite Difference Method. In *ESSCIRC 2022-IEEE 48th European Solid State Circuits Conference (ESSCIRC)*. IEEE, 353–356.

[34] Eike H Müller and Robert Scheichl. 2014. Massively parallel solvers for elliptic partial differential equations in numerical weather and climate prediction. *Quarterly Journal of the Royal Meteorological Society* 140, 685 (2014), 2608–2624.

[35] Takayuki Okimura, Teruyoshi Sasayama, Norio Takahashi, and Soichiro Ikuno. 2013. Parallelization of finite element analysis of nonlinear magnetic fields using GPU. *IEEE transactions on magnetics* 49, 5 (2013), 1557–1560.

[36] M Necati Özişik, Helcio RB Orlande, Marcelo J Colaço, and Renato M Cotta. 2017. *Finite difference methods in heat transfer*. CRC press.

[37] Zhaoxia Pu and Eugenia Kalnay. 2019. Numerical weather prediction basics: Models, numerical methods, and data assimilation. *Handbook of hydrometeorological ensemble forecasting* (2019), 67–97.

[38] Tor Steinar Schei. 1997. A finite-difference method for linearization in nonlinear estimation algorithms. *Automatica* 33, 11 (1997), 2053–2058.

[39] AMBA Tveito and Are Magnus Bruaset. 2006. *Numerical solution of partial differential equations on parallel computers*. Springer.

[40] Henk A Van der Vorst. 1992. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on scientific and Statistical Computing* 13, 2 (1992), 631–644.

[41] Hasitha Muthumala Waidyasooriya and Masanori Hariyama. 2016. FPGA-based deep-pipelined architecture for FDTD acceleration using OpenCL. In *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*. IEEE, 1–6.