FAWS: FPGA Acceleration of Large-Scale Wave Simulations

Dimitrios Gourounas*, Bagus Hanindhito*, Arash Fathi†, Dimitar Trenev†, Lizy K. John* and Andreas Gerstlauer*
* The University of Texas at Austin, Austin, TX, USA
† ExxonMobil Technology and Engineering, Annandale, NJ, USA
{dimitrisgrn,bagus}@utexas.edu, {arash.fathi,dimitar.trenev}@exxonmobil.com, {ljohn,gerstl}@utexas.edu

***Abstract*—Efficiently solving large-scale scientific problems described by partial differential equations (PDEs) is a critical task in high-performance computing. Application-specific hardware design is a well-known solution, but the wide range of kernels makes it infeasible to provision supercomputers with accelerators for all applications. This makes reconfigurable platforms a promising direction. In this work, we focus on wave simulations using discontinuous Galerkin solvers, as an important class of PDE applications. Existing work using FPGAs is limited to accelerating specific kernels or small problems that fit into FPGA BRAM. We present `FAWS`, a generic and configurable architecture for large-scale accelerated wave simulation problems running on FPGAs out of DRAM. `FAWS` exploits fine- and coarse-grain parallelism using a scalable array of application-specific processing cores, and incorporates novel dataflow optimizations, including prefetching, kernel fusion, and memory layout optimizations to minimize data transfers and maximize DRAM bandwidth utilization.**

We demonstrate `FAWS` on the simulation of elastic wave equations. Results show that a single FPGA core achieves 2.2x higher performance than 24 Xeon cores with 18.64x better energy efficiency, when given ∼1.94x less peak DRAM bandwidth. Scaling to the same peak DRAM bandwidth, an FPGA is 4.27x and 1.96x faster than 24 CPU cores and an Nvidia P100 GPU, with 31.33x and 5.29x better efficiency, respectively.

I. Introduction

Many scientific computing problems are modeled by partial differential equations (PDEs). The size and complexity of these models necessitate their numerical solution on high-performance computing (HPC) systems, where increasing model fidelity, reducing time-to-solution, and improving computational performance is often desired. These requirements motivate designing more powerful and more efficient computing systems, along with algorithms that perform well on them.

In this paper, we consider wave simulations as an important class of PDEs that fall under the category of hyperbolic PDEs, characterized by their local communication patterns. We demonstrate our approach on a member of this class, the elastic wave equation [1], which has applications in seismic hazard mitigation [2], exploration geophysics [3], [4], medical imaging [5], nondestructive testing [6], physical oceanography [7], and defense [8], [9], among others. We explore high-performance and scalable system designs that use discontinuous Galerkin (dG) methods [10] to numerically solve PDEs in a discretized manner. Such approaches are attractive in many applications [11] due to their appealing characteristics in modern hardware, such as locality, lower communication, and ease of parallelization [12]. Such problems are massively parallel, but are typically memory-bound when given enough compute resources. As such, this domain represents an example of memory-bound applications characterized by unique mesh-type data dependencies.

Traditionally, supercomputing clusters utilizing many-core CPUs are used to solve such problems. However, CPUs have a high complexity and are inefficient in achieving peak utilization. GPUs can better exploit available parallelism, but still come with inherent overheads of programmability. Hardware specialization is well known to significantly improve both performance and energy efficiency. However, unlike in other fields, such as machine learning, the large diversity of PDE kernels makes reconfigurable hardware platforms attractive for such HPC problems. Prior work has explored wave simulations using dG methods on FPGAs, but only for specific applications where selected kernels are accelerated [13] or for small problem sizes that fit entirely in the FPGA's BRAMs [14].

In this work, we propose `FAWS`, a generic and configurable FPGA architecture for large-scale accelerated wave simulations using dG methods. `FAWS` is composed as a scalable array of reconfigurable element processor (EP) cores to exploit and trade off fine- versus coarse-grain parallelism. `FAWS` specifically applies a range of optimizations to improve memory behavior and attainable DRAM bandwidth. Our main contributions include:

- We explore the design space of EP micro-architectures to demonstrate performance-efficiency tradeoffs in execution of basic PDE kernels.
- We apply a range of dataflow optimizations that employ kernel fusion to improve data movement and locality, as well as element scheduling and pipelining to exploit available parallelism.
- We further propose novel memory layout and memory channel parallelism optimizations to improve memory access patterns and maximize DRAM bandwidth utilization.
- We present a scalable wave simulation system architecture composed as a configurable array of EPs.
- `FAWS` is evaluated on an elastic wave simulation application running on an Intel Stratix 10 FPGA. Results show that, when scaled to the same bandwidth, a 2-EP and 11-EP `FAWS` design outperforms a 24-core Xeon CPU and an Nvidia P100 GPU, respectively, by 327% and 96%, with 31.33x and 5.29x better energy efficiency.

DOI 10.1109/ASAP57973.2023.00025

II. Related Work

Various software libraries [15], [16] have been proposed for mesh-based stencil computations on multi-node CPU and GPU clusters. Such libraries allow for mesh/graph generation, parallel kernel executions and optimizations such as loop fusion and tiling to maximize data reuse and parallelism [17], [18]. However, these works do not support FPGA implementations.

Wave simulations using dG methods have been widely studied on traditional supercomputers using many-core CPU nodes [19]. More recently, various works [20] have explored GPUs for the same task. Krueger et al. [21] conducted a study to compare the energy efficiency between CPU, GPU and a general-purpose many-core design for seismic modeling problems. However, GPU implementations for our particular dG setup on general elastic wave applications are limited [22]. Some recent work [23] has examined emerging Processing-in-Memory (PIM) architectures for the solution of acoustic and elastic wave simulations, but they rely on emerging technologies not readily deployable in near-term supercomputers.

FPGAs have been studied for acceleration of stencil-based PDE solvers [24]–[26]. However, their optimizations are limited to small-size stencils. Several groups have specifically investigated acceleration of wave simulations using Finite-Element [27] or Spectral-Element [28] methods on FPGAs. However, dG-based works are still limited. Kenter et al. [13] developed an FPGA accelerator using dG with unstructured meshes and tetrahedral elements. However, this solution only accelerates selected kernels and does not map the entire simulation on-chip. In a later work [14], they ported the entire simulation onto an FPGA for a shallow-water model, but they only target small problem sizes, where the full mesh fits into the FPGA's on-chip SRAM. Tomczak et al. [29] implemented a full simulation process on FPGAs for the Navier-Stokes equation, but this falls under the category of parabolic PDEs. By contrast, we present the first configurable and scalable FPGA design for large-scale, hyperbolic wave simulation problems running out of DRAM under a dG scheme.

III. Background

Hyperbolic PDEs (e.g., acoustic, elastic, and electromagnetic waves, and Euler equations) are commonly solved using methods that discretize problems in space and time. Spatial discretization using a discontinuous Galerkin method (Fig. 1) results in a discrete form for each spatial element e where $\mathbf{u}^e$ is a vector of 3D tensors holding discrete unknown *variables* on the *nodes* of each element. We focus on problems that permit using straight-faced hexahedral elements with Gauss-Lobatto-Legendre (GLL) quadrature integration, which arise in many compute-intensive, industry-relevant problems [30]. This leads to the reduction of local computations, which decreases the time-to-solution, but increases the ratio of communication to computation. Temporal discretization, e.g. by using a low-storage Runge-Kutta time-stepping, then results in repeated updates of solution vectors $\mathbf{u}^e$ for each element in each time step as shown in Alg. 1, where $\mathbf{a}^e$ is an *auxiliary* vector, $\mathbf{M}$ is the element's *mass matrix* and α, β, and γ are scalars that

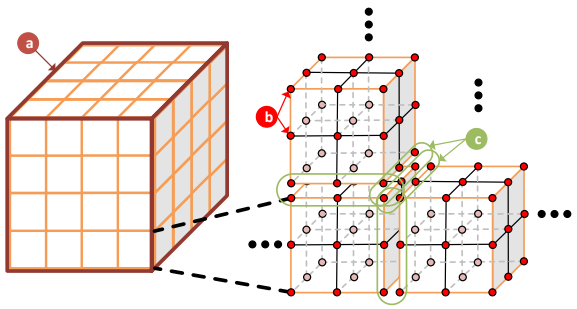

Fig. 1: Discontinuous Galerkin discretization. (a) The problem domain is represented by 3D hexahedral *elements* (mesh). (b) Each element contains $N \times N \times N (N = 3$ in this schematic) nodes, at which the unknown *variables* $\mathbf{u}^e$ are computed. The *Volume* computations take place on all of the element's nodes. (c) *Flux* computations on each face reconcile possibly discontinuous solution values among nodes on both sides.

characterize the time-stepping scheme. $\mathcal{F}, \mathcal{V}$ indicate *Flux* and *Volume* computations, respectively. The vector of 3D tensors $\mathbf{c}^e$ stores their *contributions*, and $e' \in \mathcal{N}(e)$ denotes the immediate neighbors of an element.

Volume computations determine the local contributions to unknown variables at given *node* locations within the element. Computing $\mathcal{V}(\mathbf{u}^e)$ only needs information from that element (Fig. 1(b)). By contrast, *Flux* computations reconcile discontinuities at nodes that live on the *faces* between neighboring elements. They are non-local, i.e., computing $\mathcal{F}(\mathbf{u}^e, \mathbf{u}^{e'})$ needs the corresponding nodal information of immediate neighbors (Fig. 1(c)). Using straight-faced hexahedral elements along with GLL quadrature simplifies: a) $\mathcal{V}(\mathbf{u}^e)$ into dot-product operations of a constant differentiation vector and subsets of $\mathbf{u}^e$ of size N to compute $\mathcal{O}(N^3)$ derivatives along each spatial dimension; b) $\mathcal{F}(\mathbf{u}^e, \mathbf{u}^{e'})$ into vector additions and scaling that depend on material properties; and c) $\mathbf{M}^{-1}$ into a diagonal matrix, reducing all computations to Level-1 BLAS.

In this paper, we specifically demonstrate our approach on the elastic wave equation with unknown stress $\mathcal{S} = \mathcal{S}(x, y, z, t)$ and velocity $v = v(x, y, z, t)$, which consist of six and three distinct variables, respectively. Strategies we report on herein are quite versatile. They can almost identically be applied to other hyperbolic PDEs, which are characterized by having local communication patterns. With minor adjustments, these strategies can be applied to problems discretized by other element types, such as tetrahedral elements, which are typically needed for problems that involve complex geometry, or other quadrature. These choices lead to significantly more local computations in the form of Level-2 and Level-3 BLAS operations, which then decrease the communication to computation ratio, thus making it easier to hide latency.

IV. Dataflow Optimizations

There are different ways of realizing the computations of Alg. 1, which affect locality and parallelism. In this section, we discuss the data dependencies, as well as dataflow optimizations to improve the memory behavior.

Algorithm 1: Wave simulation

1 **for** *all time steps and integration stages* **do**
2 **for** *all elements* e **do**
3 $\mathbf{c^e} = \sum_{e' \in N(e)} \mathcal{F}(\mathbf{u^e}, \mathbf{u^{e'}}) + \mathcal{V}(\mathbf{u^e})$
4 $\mathbf{a^e} = \alpha\ \mathbf{a^e} + \beta\ \mathbf{M}^{-1}\mathbf{c^e}$
5 $\mathbf{u^e} = \mathbf{u^e} + \gamma\ \mathbf{a^e}$

A. Original dataflow

Fig. 2 shows the dataflow graph corresponding to Alg. 1. The *Volume* kernel calculates the *Volume* contributions $\mathcal{V}(\mathbf{u}^e)$ on all the N^3 nodes of element e. It is independent from other elements. The *Flux* kernel runs once for each face between elements and computes the *Flux* contributions $\mathcal{F}(\mathbf{u}^e, \mathbf{u}^{e'})$ on the N^2 face nodes using the face node variables $\mathbf{u}^e$ of e and $\mathbf{u}^{e'}$ of the neighbor e'. The contributions calculated from *Volume* and *Flux* are forwarded to the *Integrate* kernel, where the variables $\mathbf{u}^e$ and auxiliaries $\mathbf{a}^e$ are updated for the next time step. This process is repeated for all elements and all stages of the Runge-Kutta time-stepping.

There are multiple possible patterns to traverse the graph. Prior CPU and GPU implementations first iterate over the *Volume* kernels of all elements. Since *Volume* kernels are independent, they can be processed in parallel. The *Flux* contributions for the face nodes of both neighboring elements are then calculated and added to the corresponding Volume contributions by iterating over all faces between elements. Different faces can be processed in parallel. However, there is pseudo-dependency when the *Flux* kernel runs on two or more faces that share nodes on a cube's edges and/or corners. A race condition occurs when they try to update shared contributions simultaneously. A synchronization of the faces is needed to avoid this dependency. Finally, the *Integrate* kernel is invoked for every element in parallel.

This dataflow, despite the parallelism it provides, incurs a large communication overhead. Specifically, inputs needed by *Volume*, *Flux* and *Integrate* kernels need to be fetched repeatedly from main memory. Additionally, the contributions computed by the *Volume* and *Flux* kernels must be stored in main memory and fetched again later when the *Integrator* runs.

B. Kernel fusion optimization

We apply a kernel fusion optimization to improve data locality and avoid repeated loading and storing of data from/to main memory. We fuse all three kernels into a larger one that fully processes each element, such that any information needed to process an element can be fetched once and used by all three internal kernels. The intermediate contributions $\mathbf{c}^e$ can be stored in separate locations $\mathbf{c}^{\mathbf{e}}_{\text{vol}}$ and $\mathbf{c}^{\mathbf{e}}_{\text{f}}$ for the *Volume* and *Flux*, respectively, and locally exchanged between kernels. The *Volume* computations can then run in parallel to the *Flux* kernels, where each updates their own sets of contributions. As before, when *Flux* runs, it also updates the contributions on the neighbor's face. A flag can be used to indicate that a neighbor does not re-run the *Flux* kernel on a face that has already been computed. Note that fusing introduces a write-after-read hazard on variable updates between *Flux* and *Integrate* kernels of neighboring elements. Not re-running already computed faces avoids this dependency. With this, three *Flux* kernels are run on average per element when using hexahedral elements.

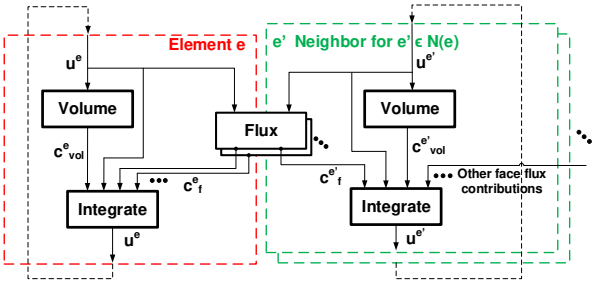

Fig. 2: Dataflow graph of Alg. 1.

C. Element scheduling

Kernel fusion increases locality and maintains a massive degree of parallelism, as multiple elements can be invoked in parallel. However, the pseudo-dependency and race condition mentioned earlier remains exposed if two elements concurrently try to update nodes on shared edges and/or corners of a common neighbor. This can be resolved by using and later aggregating different sets of *Flux* contributions for each face. Another race condition occurs if two neighboring elements try to update each other's *Flux* face contributions concurrently. Scheduling elements such that no elements processed in parallel are neighbors or share neighbors resolves these issues. We will discuss our element scheduling approach in Section V-C.

V. System Architecture

We propose a generic, scalable architecture for the numerical solution of wave simulations using dG methods. Our architecture executes a complete wave simulation (Alg. 1) in the FPGA. To perform a simulation, the host CPU generates an element mesh, downloads it into FPGA DRAM and then triggers the FPGA to start. Upon completion, the FPGA notifies the host, which then reads results from FPGA DRAM. As shown in Fig. 3, our architecture is comprised of a configurable number of instances of compute engines that implement the fused kernel, called *Element Processors* (EPs). Internally, each EP contains a *Volume*, a *Flux* and an *Integrate* kernel. The element mesh is stored in DRAM. Load and Store units are responsible for reading and writing inputs and outputs from/to DRAM into local SRAMs used as scratchpad memories within each EP. Local SRAMs store the variable $\mathbf{u}^e$, auxiliary $\mathbf{a}^e$, contribution $\mathbf{c}^e$ and mass inverse ($\mathbf{M}^{-1}$) vectors, as well as the required neighbor data. They are also used to cache some constant values that are specific to each element and take part in the *Volume* and *Flux* computations. These SRAMs are used to forward data between kernels in the EP and load/store units. The system has a main controller that enforces a proper scheduling of communications and computations.

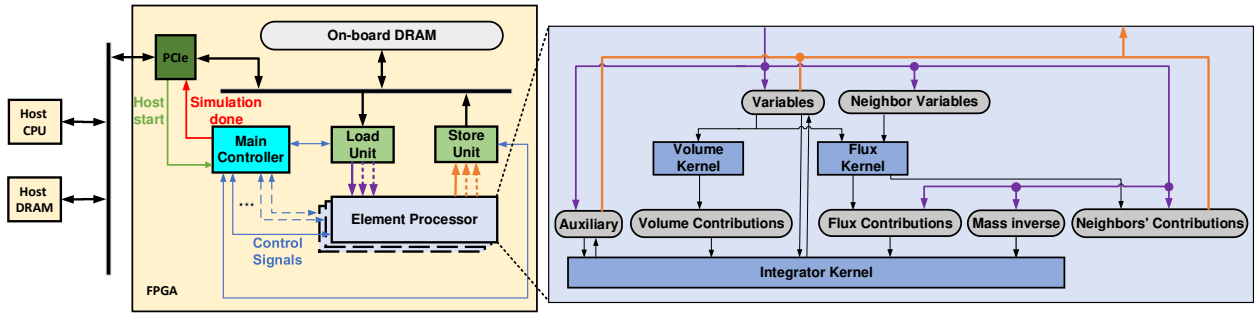

Fig. 3: FAWS architecture.

Algorithm 2: *Volume* kernel $\mathcal{V}(\mathbf{u^e})$

Inputs: Variable vector of 3D tensors $\mathbf{u}^e$ and constant differential vectors $const_dx, const_dy, const_dz$

Outputs: Contribution vector of 3D tensors $\mathbf{c}^e_{vol}$

for *all* N^3 *nodes (i,j,k)* **do**
  for *offset* $o = 0, \ldots, N-1$ **do**
    for *variable* $r = 0, \ldots, R$ **do** *some of*
      $\frac{\partial u^e_{r,i,j,k}}{\partial x} \mathrel{+}= u^e_{r,(i+o)\%N,j,k} * const_dx_o$
      $\frac{\partial u^e_{r,i,j,k}}{\partial y} \mathrel{+}= u^e_{r,i,(j+o)\%N,k} * const_dy_o$
      $\frac{\partial u^e_{r,i,j,k}}{\partial z} \mathrel{+}= u^e_{r,i,j,(k+o)\%N} * const_dz_o$
    end
  end
  $\forall r : c^e_{vol,r,i,j,k} = f(\nabla u^e_{0,i,j,k}, \ldots, \nabla u^e_{R,i,j,k})$
end

A. Element Processor Design

We further detail the EP-internal design. Through modifications of the computation kernels and the memory structure, our EP micro-architecture is general in supporting a wide range of applications of wave simulations. In the following, we present a design space exploration of each kernel that goes into designing an application-specific EP.

Volume: There is a large degree of fine-grain parallelism within a typical *Volume* kernel, the general structure of which is shown in Alg. 2. This kernel calculates the vector $\mathbf{c}^{\mathbf{e}}_{\mathrm{vol}}$ of 3D tensors with the *Volume* contributions for element e. The outer-most loop iterates over all the N^3 nodes in the 3D element. The inner loops calculate the variable derivatives in each direction (x, y, z) as the dot products between the variable vectors and the constant differential vectors mentioned in Section III. Not all three derivatives are needed for all the variables, which introduces some conditional executions within this loop. Finally, the *Volume* contributions of a node are computed as a simple function of the calculated derivatives.

Our goal is to maximize computational throughput while minimizing area and energy. We explore different options for exploiting *Volume* parallelism. Each variable's dot products in the inner-most loop can be processed in parallel by fully unrolling the loop. Doing so removes any conditional statements. The result is an outer loop that iterates over nodes and one inner loop that iterates over N to calculate every dot product for all the variables. This was found to always be efficient in terms of the performance-area trade-off, so we consider it as a baseline design. To further improve throughput, we can also unroll the inner loop, implementing the dot products as a set of multipliers and adder trees, or we can unroll both the inner and the outer loop as the nodes are also independent. Finally, parallelism can be attained by instantiating multiple *Volume* kernels that run on different elements concurrently. All design options are pipelined with an initiation interval (II) of 1 in their inner-most loops.

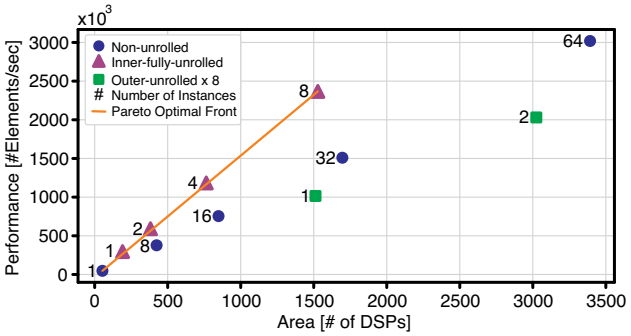

Fig. 4: *Volume* kernel's design space.

Fig. 4 shows the area-performance design space of the Volume kernel's baseline design for our elastic wave example, with $N = 8$. Similar behavior is observed for different values of N. To measure area, we use the total number of DSP blocks used for floating-point operations on our FPGA. The design options also affect the number of BRAMs and Look-Up-Tables (LUTs), but similar conclusions can be drawn for these resources. Overall, we see that the design with the optimal area-performance trade-off is the one with the inner loop fully unrolled. In other words, when designing EP micro-architectures, it is always more efficient to exploit fine-grain operation-level parallelism compared to instantiating multiple simpler kernel-level processors. At the same time, exposing too much parallelism when unrolling the outer loop offers diminishing returns, due to the initial depth of the kernel's datapath. To evaluate such effects in the full system context, we implemented two designs for our example application that have a single *Volume* kernel per EP with the inner loop rolled and fully unrolled, respectively.

Algorithm 3: *Flux* kernel $\mathcal{F}(\mathbf{u^e}, \mathbf{u^{e'}})$

Inputs: $\mathbf{u}^{e(f1)}, \mathbf{u}^{e'(f2)}$: 2D variable tensors of element e and neighbor $e' \in \mathcal{N}(e)$ on faces $f1$ and $f2$
Outputs: Flux vectors of 2D tensors $\mathbf{c}_{\mathrm{f}}^{e(f1)}$, $\mathbf{c}_{\mathrm{f}}^{e'(f2)}$

```
for all N^2 face nodes (i,j) do
    for variable r = 0, ..., R do
        c^{e(f1)}_{f,r,i,j}, c^{e'(f2)}_{f,r,i,j} = g(u^{e(f1)}_{w,i,j}, u^{e'(f2)}_{w,i,j}, w = 0, ..., R)
    end
end
```

Flux: The Flux kernel calculates the flux contribution vectors of 2D tensors $\mathbf{c}_{\mathrm{f}}^{\mathbf{e}}$ and $\mathbf{c}_{\mathrm{f}}^{\mathbf{e}'}$ for elements *e* and *e'* on one shared face. Its general structure is shown in Alg. 3. The outer loop iterates over all the N^2 face nodes, while the inner loop iterates over the variables to calculate the contributions of both elements. As discussed in Section III, the actual face *Flux* computation uses vector additions and scaling as a function of the face node variables of both elements. The inner loop can be fully unrolled, as each variable can be processed in parallel, resulting in a single loop that iterates over the face nodes. To increase throughput, one option is to unroll this loop as well. An alternative is to have multiple kernels processing separate faces concurrently. As mentioned in Section IV, for hexahedral elements, *Flux* runs three times on average per element. Crucially, it does not make sense for the *Flux* kernels to be much faster than the *Volume*. In an optimal design, *Flux* and *Volume* computations are balanced to run in parallel for the same amount of time. In our example application, a single *Flux* kernel with the outer loop rolled and the inner loop fully unrolled was found to be enough to achieve this balance. This leads to a single pipelined loop with $II = 1$.

Integrate: The *Integrate* kernel consists of a single pipelined loop ($II = 1$) that iterates over all nodes in an element and updates all unknown variables $\mathbf{u}^e$ and auxiliaries $\mathbf{a}^e$ based on the contributions $\mathbf{c}^e$. It can be accelerated by unrolling the loop, offering diminishing returns due to the latency of the datapath, so we chose not to in our designs.

Memory Structure: The variable ($\mathbf{u}^e$), auxiliary ($\mathbf{a}^e$), contribution ($\mathbf{c}^e$) and mass inverse ($\mathbf{M}^{-1}$) quantities for the element nodes, as well as the neighbor data and element constants, are each stored in their own SRAMs. Each SRAM is implemented by grouping together multiple BRAMs. As some of the variable arrays are accessed multiple times per clock cycle, depending on BRAM limitations in the target FPGA, such arrays need to be replicated to provide more access ports. Table I shows the replication factor of the nine variables in our setup for the elastic wave problem (six for stress and three for velocity) depending on whether the *Volume* kernel's inner loop is unrolled or not. Loop unrolling increases the BRAM accesses per clock cycle and hence the replication factor.

B. Load/Store Units

Load/Store units are responsible for transferring data between the EP scratchpad memories and DRAM. To efficiently utilize available DRAM bandwidth, careful design of memory transactions and the data layout in DRAM is required. In our FPGA, the logic operates at a $4\ ns$ clock period and the port width is up to 1024 bits, which translates to a peak bandwidth of 29.8GB/s per interface. Additionally, with a 512-bit wide interface, we have a total of 16 floating-point values available every time the DRAM controller responds with data. In order to not stall before issuing following transactions, these values need to be stored in the SRAMs at a speed that is at least equal to that of the DRAM.

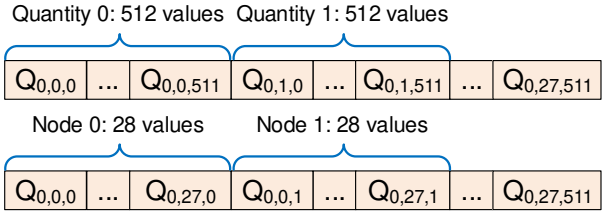

Fig. 5: Original (Top) and modified (Bottom) DRAM layout.

TABLE I: Stress (S) and velocity (V) replication in memory.

Variables	Non-unrolled Volume	Unrolled Volume
S0, S1, S2	1	5
S3, S4, S5	2	9
V0, V1, V2	2	13

Fig. 5 (Top) shows the original data layout for our example application, which has 512 nodes per element and 28 quantities (32-bit long each) per node (variables, auxiliaries, contributions and mass inverse). We use the notation $Q_{e,q,n}$ to refer to element *e*, quantity *q* and node *n*. In this layout, the element data is stored variable by variable. When using this DRAM data layout and a single 512-bit wide DRAM port, all 16 values that are fetched from a DRAM transaction must be stored on the same SRAM, as they all belong to the same quantity. As a result, each individual SRAM must be designed to have 512-bit wide ports, which increases FPGA BRAM usage. Finally, in a naive baseline design, the constants of an element (10 in total) are fetched to a local SRAM prior to each element's execution, neighbor data is fetched even for faces that have already been computed, and all of each neighbor's nodes are always fetched. In the following, we describe optimizations that we applied to improve memory performance, bandwidth utilization and resource usage.

Memory optimizations: We first optimize the load/store units to skip loading/storing neighbors for faces that are already computed, and we only fetch the one face of each neighbor that we need. This increases the irregularity of the DRAM access patterns, as the nodes of a face are not always stored consecutively in DRAM. To increase regularity, we further rearrange the memory layout. As shown in Fig. 5 (Bottom), in the optimized layout, we store data per node rather than per variable. With a single 1024-bit wide interface, we can then issue all 28 transactions in one cycle. Each of the 28 values goes to a different SRAM, which reduces the bit-width of the ports to 32, significantly lowering BRAM and LUT usage. In addition, neighbor face node accesses

	Nodes									
Face 0	0	8	16	...				488	496	504
Face 1	7	15	23	...				495	503	511
Face 2	0	...	7	64	...	71	...	448	...	455
Face 3	56	...	63	120	...	127	...	504	...	511
Face 4	0	1	2	...				61	62	63
Face 5	448	449	450	...				509	510	511

Fig. 6: Node IDs of each face.

also have better regularity, since all the quantities of a face node are consecutive. Finally, the constants of all the elements are prefetched prior to execution and cached in a dedicated SRAM, improving regularity and reducing memory traffic.

Multi-channel DRAM: We extend our Load/Store units to utilize multiple memory channels (4 in our FPGA board) and increase bandwidth. We assume that each channel accesses its own DRAM address space (e.g., a dedicated DIMM per channel) and that all element data is equally partitioned across channels. However, access patterns using the aforementioned DRAM layouts do not guarantee that all channels will be equally used when fetching different neighbor faces. Figure 6 shows the node IDs that belong to each face of an element. Using the original mapping of nodes, as shown in Figure 7 (top), all the nodes of faces 0 (red) and 1 (green) belong exclusively to DIMMs 0 and 3, respectively. This problem would lead to poor bandwidth utilization when fetching neighbor data, as well as introduce a complex control mechanism on the Load/Store units. To avoid this issue, we implement a skewing mechanism similar to [31] for interleaving nodes across different DIMMs, as shown in Figure 7 (bottom) for our example of 4 DIMMs. In this example with $N = 8$, node j is mapped to DIMM $(j + \lfloor j/8 \rfloor)\%C$, where C is the number of DRAM channels. This skewing formula can be applied to any number of channels that is a power of 2 and less than or equal to 8. For 16 and 32 channels, the formula changes to $(j + 2 * \lfloor j/16 \rfloor)\%16$ and $(j + 4 * \lfloor j/32 \rfloor)\%32$, respectively. If the number of channels is not a power of 2, the mapping will be unbalanced. This mapping enables equal utilization of all memory channels, leading to higher bandwidth and a simpler control logic. The Load/Store units and compute kernels are modified to utilize a flexible mapping table to access the correct node in the SRAMs, where different mapping schemes can be realized by changing mapping tables.

C. Main Controller

The main controller implements a dataflow-driven schedule of computations, which keeps track of proper kernel ordering and handles dependencies on shared data. We apply optimizations, such as prefetching, pipelining and dependency-aware element scheduling.

Pipelining: The system controller realizes a dynamic, data-driven schedule that allows for inherently flexible, self-arranged pipelining and overlapping of computation and communication by launching computation kernels and load/store operations as soon as their data is ready. Fig. 8 shows a resulting memory-bound schedule in which prefetching and loading of the next element, as well as storing back of

	Nodes – Original mapping											
DIMM 0	0	4	8	12	16	20	24	28	32	36	40	...
DIMM 1	1	5	9	13	17	21	25	29	33	37	41	...
DIMM 2	2	6	10	14	18	22	26	30	34	38	42	...
DIMM 3	3	7	11	15	19	23	27	31	35	39	43	...

	Nodes – Skewing											
DIMM 0	0	4	11	15	18	22	25	29	32	36	43	...
DIMM 1	1	5	8	12	19	23	26	30	33	37	40	...
DIMM 2	2	6	9	13	16	20	27	31	34	38	41	...
DIMM 3	3	7	10	14	17	21	24	28	35	39	42	...

Fig. 7: Mapping of nodes across different DIMMs using the original layout (top) and skewing (bottom).

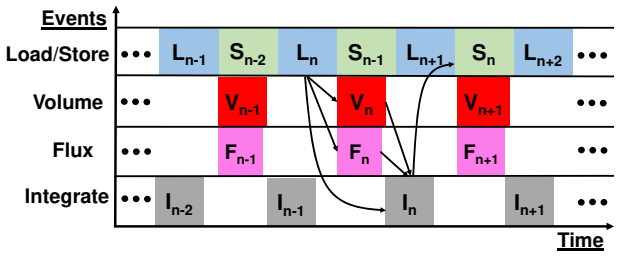

Fig. 8: System dataflow schedule.

the previous element are overlapped with *Volume*, *Flux* and *Integrate* kernels. Such a dynamic scheduler also allows the *Integrate* kernel to be overlapped with *Volume* and *Flux* in compute-bound cases. A similar pipelined schedule can be used when multiple EPs are running in parallel by extending Load/Store units to fetch and write back multiple elements at a time. To allow for such pipelining, any quantities that sit on the interface between the Load/Store units and the EPs need to be duplicated (double-buffering).

Element scheduling: A proper ordering of elements is needed to avoid the data dependencies across subsequent elements mentioned in Section IV. For example, in Fig. 8, *Load* $n+1$ must not start before *Store* $n-1$ finishes if element $n+1$ is a neighbor of element $n-1$. In this case, *Load* $n+1$ needs to stall until *Store* $n-1$ updates the *Flux* contributions of $n+1$ on the face they share. In fact, subsequent elements should also not share neighbors, as they may both try to update the *Flux* contributions on the same edge and/or vertex nodes. In our case, we implemented a simple static schedule that satisfies this requirement and avoids all dependencies.

VI. Experiments and Results

We evaluated `FAWS` on the DE10-Pro board with an Intel Stratix 10 SX 2800 FPGA running the elastic wave equation problem for different problem sizes that fit in DRAM. We compare our designs against a 24-core Intel Xeon Platinum 8160 implementation that realizes a baseline and optimized dataflow, and a GPU baseline implementation running on a Nvidia P100 with 56 SMs. No further optimizations (such as vectorization) are applied to the CPU. These platforms were chosen because they were released around the same time (with Stratix being the oldest) and they have similar technology

TABLE II: Hardware platform configurations.

	CPU	FPGA	GPU
Device	Xeon 8160	Stratix 10 SX2800	Nvidia P100
Peak Freq.	3.7GHz	1GHz	1.33GHz
On-chip memory	59.5MB	28.625MB	23.22MB
Peak FLOPS	1.4T	9.2T	9.3T
DRAM	DDR4–2667	DDR4–2667	HBM2
Peak BW	119GBps	19.2GBps/DIMM	734GBps
Technology node	14nm	14nm	16nm
Released	2017	2013	2016

TABLE III: Resource usage of system variants.

Designs		DSPs	BRAMs	ALMs
Base	**EP**	173 (3%)	4304 (36.72%)	44974 (4.82%)
	L/S	0 (0%)	240 (2.04%)	50386 (5.4%)
	I/C	0 (0%)	0 (0%)	41934 (4.5%)
M	**EP**	173 (3%)	349 (2.9%)	19655 (2%)
	L/S	0 (0%)	128 (1%)	23049 (2.4%)
	I/C	0 (0%)	0 (0%)	6220 (0.6%)
MU	**EP**	320 (5.5%)	651 (5.5%)	32045 (3.43%)
	L/S	0 (0%)	143 (1.22%)	27143 (2.9%)
	I/C	0 (0%)	0 (0%)	7385 (0.79%)
MU4	**EP**	320 (5.5%)	1160 (9.9%)	36312 (3.89%)
	L/S	0 (0%)	468 (3.99%)	81790 (8.76%)
	I/C	0 (0%)	0 (0%)	17562 (1.88%)
Controller		0 (0%)	0 (0%)	95 (∼0%)
Other		0 (0%)	1170 (10%)	76316 (8.18%)

nodes as our FPGA. Table II summarizes the hardware specifications of our FPGA, CPU and GPU platforms.

Elastic wave application: The element mesh is partitioned into a number of elements with 512 nodes each (N=8) and nine variables per node. Also, there are nine corresponding auxiliaries and contributions, and one mass inverse value for each node. Single-precision (32-bit) floating-point format was used for all the values. This results in a ∼56kB element size. We tested for problem sizes of 512, 4096 and 32768 elements. All designs use *p4est* [16] to generate the mesh.

System variants: We compare a baseline load/store design with a rolled *Volume* kernel (Base) against variants with memory optimizations (M), *Volume* loop unrolling (U) and multi-channel (4) optimizations in MU and MU4 combinations.

Hardware design: We used Intel Quartus High-Level-Synthesis (HLS) to design the EP's kernels and load/store units. The main controller was designed in Verilog. Our implemented system instantiates a single EP. In this work, we target a system with 1 and 4 DIMMs, each with its own memory channel. The peak bandwidth of the 3 of the 4 DIMMs is 19.2GB/s, while the fourth DIMM has a peak bandwidth of 15.35GB/s. Since data is equally balanced across all 4 DIMMs and all 4 are accessed in parallel, frequency is limited by the slowest DIMM, leading to a total peak bandwidth of 61.4GB/s. Up to 2 EPs were found to be enough to saturate the attainable bandwidth for all designs.

FPGA synthesis: We used Intel Quartus Prime 2019.3 to synthesize all designs. Floating-point operations are mapped on hard DSP blocks with floating point capabilities. The local

TABLE IV: DRAM traffic and bandwidth of each variant.

Variant	Base	M	MU	MU4
Traffic (kB/El.)	434	130	130	130
BW (GB/s)	4.52	7.8	10.6	21.2

TABLE V: Average latency (in cycles) of compute kernels.

Kernel	`FAWS-MU`	FAWS-M/FAWS-Base
Volume	772	4240
Flux	365	365
Integrate	610	610

EP SRAMs are implemented using M20K Memory Blocks as BRAMs. We use a 250MHz clock for our designs.

Power measurements: We used the Quartus Power Analyzer to measure the FPGA's core dynamic and static power, RAPL counters with the linux perf system for the CPU's entire package power, which includes the 24 cores and the cache, and the nvidia-smi dmon tool for GPU power.

A. Resource Utilization

Table III summarizes the resource utilization of the system for Base, M, MU and MU4 designs. Resources are broken down into EP (including the three kernels and the SRAMs), load/store (L/S) units, interconnect (I/C) and main controller usage. "Other" indicates the total resources required for fixed components, such as PCIe, DRAM controllers, clock-crossing bridges and the element constant SRAM for a 32,768-element problem size. Overall, M requires significantly fewer BRAMs and ALMs for the EPs and the interconnects than the Base design. Unrolling the *Volume* inner loop increases EP DSP and BRAM usage by 85% and 86%. Multi-channel parallelism increases the logic required by the L/S units and the interconnects. BRAM usage is also increased since we need 64-bit wide SRAM ports to store all 4 values fetched per transaction in one cycle and simplify the control logic of the L/S units.

B. Performance and Efficiency

Memory performance: Table IV shows the total amount of data transferred from/to DRAM per element and the bandwidth achieved by each of the variants. Memory layout optimizations increase the achieved bandwidth by 72% and reduce memory traffic by ∼70% compared to Base. However, `FAWS`-M is compute-bound, which limits achievable memory performance. `FAWS`-MU renders the design memory-bound again, where bandwidth is increased by 134% over Base. Channel parallelism further increases bandwidth by 2x. As mentioned before, the maximum bandwidth increase when using all 4 DIMMs is ∼3.2x compared to a single DIMM. Our 4 DIMM design yields less speedup (2x) than the theoretical maximum (3.2x) since the transaction size per channel is reduced. To mitigate this issue, we plan to implement an approach that loads elements in larger batches in the future.

Performance: Table V shows the execution time of each kernel in clock cycles for all designs. Since the Flux kernel's runtime depends on the number of faces it will process, we show its 3-face average runtime. Fig. 9 shows performance comparison between `FAWS` variants and the CPU (left), as well

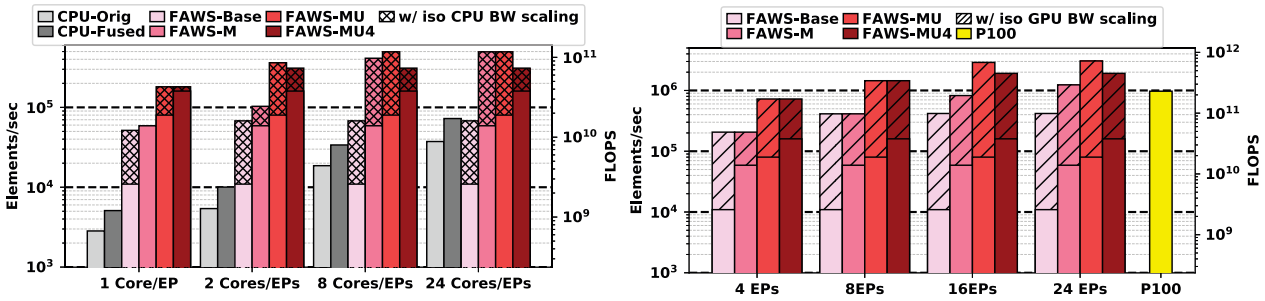

Fig. 9: FAWS performance comparison to Xeon (left) and P100 (right).

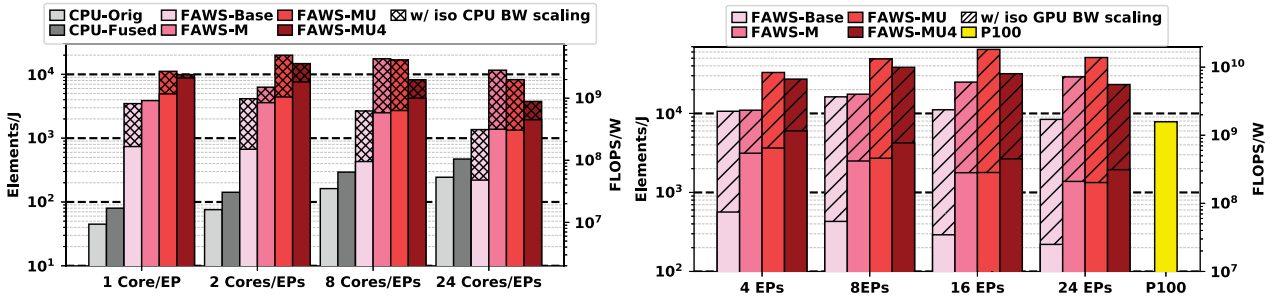

Fig. 10: FAWS variants energy efficiency comparison to Xeon (left) and P100 (right).

as the GPU (right). We used a performance model to estimate performance for designs with more than 1 EP and FPGA bandwidth that matches peak CPU or GPU bandwidth, assuming the same attainable percentage for regular and irregular accesses. CPU measurements include the original and kernel-fusion implementations with up to 24 cores. Measurements correspond to a 32768-element problem size, but a similar behavior is observed for other sizes. A single FAWS-M EP achieves the same performance as 16 CPU cores, but is compute-bound. A single FAWS-MU EP is 36% faster than a FAWS-M EP and outperforms the 8- and 24-core CPU by 136% and 10%, respectively. Memory channel parallelism further increases performance by 100%, outperforming the 24-core CPU by 120%. 2 FAWS-MU4 cores are enough to achieve 327% higher performance than 24 CPU cores. Due to the imperfect DRAM channel scaling, we observe that FAWS-MU4 achieves lower performance than an ideally scaled FAWS-MU when scaling for iso-CPU/-GPU bandwidth. Compared to the GPU, 11 FAWS-MU4 cores are needed to saturate bandwidth and outperform the P100 by 96%.

Efficiency: Fig. 10 compares the energy efficiency of LAWS variants to the CPU (left), and the GPU (right). Results are shown for the achieved and scaled-up efficiency when increasing EP counts and matching the CPU's and GPU's peak DRAM bandwidths. We use a proportional power model to scale FAWS power with the number of EPs. Peak efficiency of a FAWS design is reached at EP counts that saturate bandwidth when the communication and computation times are equal. Higher EP counts are memory-bound and hence need to stall, which limits efficiency. Lower EP counts do not utilize available resources efficiently. As can be seen from Table III, one EP uses a very small portion of the FPGA's resources. This leads to high leakage power, which would be reduced if a smaller FPGA was used. Overall, a single-EP FAWS-MU design achieves 16.8x and 10.48x higher energy efficiency than 8 and 24 CPU cores, respectively. A single-EP FAWS-MU4 design achieves 29.88x and 18.64x higher energy efficiency than the 8- and 24-core CPU, respectively. Savings increase to 31.33x over the 24-core CPU with 2 FAWS-MU4 EPs when scaling to iso CPU bandwidth. Compared to the P100, a FAWS-MU4 with 11 EPs achieves 5.3x better efficiency when scaled to the same bandwidth. As observed before, FAWS-MU4 does not scale perfectly, causing its best possible energy efficiency to be slightly lower than that of FAWS-MU with ideal scaling for peak bandwidth.

VII. Summary and Conclusions

In this work, we presented FAWS, an efficient, scalable and configurable FPGA architecture for accelerating wave simulations using the discontinuous Galerkin method. Our architecture consists of a scalable array of custom element processors (EPs) and incorporates a range of novel dataflow and memory optimizations. Results of applying our design to elastic wave simulations show that with 1.94x less peak DRAM bandwidth than the CPU, a single EP achieves 2.2x higher performance than 24 CPU cores at 18.64x higher energy efficiency. When scaling to equivalent peak bandwidth, 2 and 11 EPs can outperform a 24-core CPU and a 56-SM GPU by 4.27x and 1.96x with 31.33x and 5.29x higher efficiency, respectively. In the future, we aim to explore additional memory optimizations, such as tiling, using High-Bandwidth-Memory (HBM) and multi-FPGA implementations.

ACKNOWLEDGMENTS

This research was supported by ExxonMobil Technology and Engineering Company, agreement no. EM10480.36. Any opinions, findings, conclusions or recommendations are those of the authors and not of the sponsors.

REFERENCES

[1] B. Poursartip, A. Fathi, and J. L. Tassoulas, "Large-scale simulation of seismic wave motion: A review," *Soil Dynamics and Earthquake Engineering*, vol. 129, p. 105909, 2020.

[2] A. Fathi, B. Poursartip, K. H. Stokoe II, and L. F. Kallivokas, "Three-dimensional P- and S-wave velocity profiling of geotechnical sites using full-waveform inversion driven by field data," *Soil Dynamics and Earthquake Engineering*, vol. 87, pp. 63–81, 2016.

[3] L. Kallivokas, A. Fathi, S. Kucukcoban, K. Stokoe, J. Bielak, and O. Ghattas, "Site characterization using full waveform inversion," *Soil Dynamics and Earthquake Engineering*, vol. 47, pp. 62–82, 2013.

[4] M.-D. Lacasse, L. White, H. Denli, and L. Qiu, "Full-wavefield inversion: An extreme-scale PDE-constrained optimization problem," in *Frontiers in PDE-Constrained Optimization*, H. Antil, D. P. Kouri, M.-D. Lacasse, and D. Ridzal, Eds. Springer, 2018, pp. 205–255.

[5] L. Guasch, O. C. Agudo, M.-X. Tang, P. Nachev, and M. Warner, "Full-waveform inversion imaging of the human brain," *npj Digital Medicine*, vol. 3, pp. 1 – 12, 2020.

[6] H. Chen, M. Zhou, S. Gan, X. Nie, B. Xu, and Y. Mo, "Review of wave method-based non-destructive testing for steel-concrete composite structures: Multiscale simulation and multi-physics coupling analysis," *Construction and Building Materials*, vol. 302, p. 123832, 2021.

[7] T. Duda, J. Bonnel, E. Coelho, and K. Heaney, "Computational Acoustics in Oceanography: The Research Roles of Sound Field Simulations," *Acoustics Today*, vol. 15, pp. 28–37, 2019.

[8] S. Hong, N. Vlahopoulos, R. M. Mantey Jr, and D. J. Gorsich, "A computational approach for evaluating the probability of acoustic detection of a military vehicle," in *Targets and Backgrounds X: Characterization and Representation*, 2004.

[9] K.-H. Barth, "The politics of seismology: Nuclear testing, arms control, and the transformation of a discipline," *Social Studies of Science*, vol. 33, pp. 743–781, 2003.

[10] J. Hesthaven and T. Warburton, *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*. Springer, 2010.

[11] D. Arndt, N. Fehn, G. Kanschat, K. Kormann, M. Kronbichler, P. Munch, W. A. Wall, and J. Witte, "ExaDG: High-order discontinuous Galerkin for the exa-scale," in *Software for exascale computing-SPPEXA 2016-2019*, H.-J. Bungartz, S. Reiz, B. Uekermann, P. Neumann, and W. E. Nagel, Eds. Springer, 2020, pp. 189–224.

[12] L. C. Wilcox, G. Stadler, C. Burstedde, and O. Ghattas, "A high-order discontinuous Galerkin method for wave propagation through coupled elastic–acoustic media," *Journal of Computational Physics*, vol. 229, pp. 9373–9396, 2010.

[13] T. Kenter, G. Mahale, S. Alhaddad, Y. Grynko, C. Schmitt, A. Afzal, F. Hannig, J. Förstner, and C. Plessl, "OpenCL-based FPGA design to accelerate the nodal discontinuous Galerkin method for unstructured meshes," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018.

[14] T. Kenter, A. Shambhu, S. Faghih-Naini, and V. Aizinger, "Algorithm-hardware co-design of a discontinuous Galerkin shallow-water model for a dataflow architecture on FPGA," in *Platform for Advanced Scientific Computing Conference (PASC)*, 2021.

[15] F. Rathgeber, G. R. Markall, L. Mitchell, N. Loriant, D. A. Ham, C. Bertolli, and P. H. Kelly, "PyOP2: A high-level framework for performance-portable simulations on unstructured meshes," in *SC Companion: High Performance Computing, Networking Storage and Analysis (SCC)*, 2012.

[16] C. Burstedde, L. C. Wilcox, and O. Ghattas, "p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees," *SIAM Journal on Scientific Computing*, vol. 33, pp. 1103–1133, 2011.

[17] V. Bandishti, I. Pananilath, and U. Bondhugula, "Tiling stencil computations to maximize parallelism," in *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2012.

[18] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective Automatic Parallelization of Stencil Computations," in *Programming Language Design and Implementation (PLDI)*, 2007.

[19] A. Heinecke, A. Breuer, S. Rettenberger, M. Bader, A.-A. Gabriel, C. Pelties, A. Bode, W. Barth, X.-K. Liao, K. Vaidyanathan *et al.*, "Petascale high order dynamic rupture earthquake simulations on heterogeneous supercomputers," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[20] R. Gandham, D. Medina, and T. Warburton, "GPU accelerated discontinuous Galerkin methods for shallow water equations," *Communications in Computational Physics*, vol. 18, pp. 37–64, 2015.

[21] J. Krueger, D. Donofrio, J. Shalf, M. Mohiyuddin, S. Williams, L. Oliker, and F.-J. Pfreund, "Hardware/software co-design for energy-efficient seismic modeling," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[22] B. Hanindhito, D. Gourounas, A. Fathi, D. Trenev, A. Gerstlauer, and L. K. John, "GAPS: GPU-acceleration of PDE solvers for wave simulation," in *International Conference on Supercomputing (ICS)*, 2022.

[23] B. Hanindhito, R. Li, D. Gourounas, A. Fathi, K. Govil, D. Trenev, A. Gerstlauer, and L. John, "Wave-PIM: Accelerating Wave Simulation Using Processing-in-Memory," in *International Conference on Parallel Processing (ICPP)*, 2021.

[24] R. Cattaneo, G. Natale, C. Sicignano, D. Sciuto, and M. D. Santambrogio, "On how to accelerate iterative stencil loops: a scalable streaming-based approach," *ACM Transactions on Architecture and Code Optimization*, vol. 12, pp. 1–26, 2015.

[25] V. Rana, I. Beretta, F. Bruschi, A. A. Nacci, D. Atienza, and D. Sciuto, "Efficient hardware design of iterative stencil loops," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, pp. 2018–2031, 2016.

[26] M. Koraei, O. Fatemi, and M. Jahre, "DCMI: A scalable strategy for accelerating iterative stencil loops on FPGAs," *ACM Transactions on Architecture and Code Optimization*, vol. 16, pp. 1–24, 2019.

[27] C. He, "Numerical solutions of differential equations on FPGA-enhanced computers," Ph.D. dissertation, Texas A & M University, 2010.

[28] M. Karp, A. Podobas, N. Jansson, T. Kenter, C. Plessl, P. Schlatter, and S. Markidis, "High-performance spectral element methods on field-programmable gate arrays: implementation, evaluation, and future projection," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2021.

[29] T. Tomczak, M. Księżyk, M. Marek, J. Hanke, and M. Kostur, "Parallel Accelerator for Discontinuous Galerkin Method for Navier-Stokes Equations," in *International Conference on Systems Engineering, (ICSEng)*, 2021.

[30] A. Fathi, B. Poursartip, and L. F. Kallivokas, "Time-domain hybrid formulations for wave simulations in three-dimensional PML-truncated heterogeneous media," *International Journal for Numerical Methods in Engineering*, vol. 101, pp. 165–198, 2015.

[31] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Transactions on Computers*, vol. 100, pp. 1145–1155, 1975.