HighWave: Large-Scale High-Bandwidth Wave Simulations on FPGAs

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

***Abstract*—Wave simulations are an important task in high-performance computing across various fields. Wave problems represented as partial differential equations (PDEs) are massively parallel and thus well-suited for acceleration. FPGAs are an attractive solution due to their reconfigurability, allowing application-specific optimizations of various wave equations. However, memory bandwidth eventually becomes a bottleneck, where effective utilization of High-Bandwidth-Memory (HBM) or similar technology is essential in achieving scalable performance. GPUs rely on significant hardware resources to manage memory traffic. By contrast, FPGAs can specialize memory systems and optimize access patterns to achieve higher utilization on a lower power budget, but this requires careful design optimizations.**

This work introduces `HighWave`, a scalable FPGA-based spatial architecture that efficiently leverages HBM in acceleration of wave simulations. We focus on wave solvers that use discontinuous Galerkin schemes with Gauss-Lobatto-Legendre quadrature points, characterized by low arithmetic intensity and complex access patterns. `HighWave` integrates configurable processing cores and applies memory optimizations that maximize data reuse, minimize access irregularities, and ensure load balancing, thereby achieving high HBM bandwidth utilization. Unlike prior works for acceleration of wave simulations on FPGAs that lack HBM integration, `HighWave` provides better scalability for cluster-level deployment and supports a wide range of problems.

We evaluate `HighWave` on the simulation of elastic and acoustic wave equations using an Intel Stratix 10 MX FPGA. Despite the high irregularity of data access patterns, `HighWave` achieves up to 72.2% utilization of peak HBM bandwidth. When normalizing for equivalent peak bandwidth, it delivers up to 48% higher performance than state-of-the-art GPU implementations on an Nvidia V100, with up to 2.84× higher energy efficiency.

I. Introduction

Wave simulations are an integral component of important applications, such as hydrocarbon exploration [1], finding safe spaces for CO_2 sequestration [2], seismic hazard mitigation [3], acoustic modeling [4], aerospace engineering [5] and the defense sector [6], [7]. Increased interest to improve reliability and account for uncertainty in such applications necessitate simulations on finer grids and at larger scales, calling for significantly more computational resources. Wave problems belong to the broader class of hyperbolic partial differential equations (PDEs), characterized by their local communication patterns. While most such simulations rely on finite difference or sometimes spectral element methods (FDMs or SEMs) for wave discretization, there is considerable interest in exploring discontinuous Galerkin (dG) methods [8], due to their lower communication requirements and higher accuracy when sharp interfaces are present, e.g., when modeling fluid-solid interface at the seafloor in offshore hydrocarbon exploration applications. In particular, dG-based schemes using structured hexahedral elements and Gauss-Lobatto-Legendre (GLL) quadrature points limit computations to vector operations (Level-1 BLAS). This choice reduces overall wallclock time, making such dG schemes competitive with FDM, but it also increases the communication-to-computation ratio.

Such applications are characterized by low arithmetic intensity and complex memory access patterns, thus quickly becoming memory-bound. GPUs have been shown to outperform CPUs by orders of magnitude [9], due to their ability to extract available ample parallelism coupled with utilization of HBM. However, effective utilization of memory parallelism and bandwidth is non-trivial. GPUs incur significant hardware overhead that greatly increases power consumption without ensuring optimal HBM utilization. Hardware specialization can overcome these limitations, making FPGAs a promising alternative, but to the best of our knowledge, no prior work has designed FPGA accelerators for dG solvers utilizing HBM.

In this paper, we propose `HighWave`, the first HBM-enabled FPGA accelerator for large-scale wave simulations using a dG-based numerical solver. `HighWave` is a spatial architecture consisting of an array of configurable, highly-efficient processing cores referred to as element processors (EPs). It leverages a custom memory hierarchy and employs data movement and locality optimizations that maximize attainable HBM bandwidth. Our architecture is general and configurable to support a wide range of dG problems. With reasonable modifications, it can also support FDM and SEM, tackling a wider range of solvers for hyperbolic PDEs. We showcase our architecture on two case studies, the elastic and acoustic wave equations, which have important practical applications, such as exploration geophysics [10], [11], and seismic hazard mitigation [3], [12].

Our main contributions are summarized below:

- We propose `HighWave` as a scalable HBM-enabled architecture targeting wave simulations. `HighWave` consists of a spatial array of reconfigurable EPs combined with a specialized on-chip memory hierarchy and custom load/store (LS) memory access units.
- `HighWave` employs a range of data movement and locality optimizations that minimize irregular HBM accesses and achieve efficient load balancing across all HBM channels in order to maximize HBM bandwidth with minimum hardware overhead.

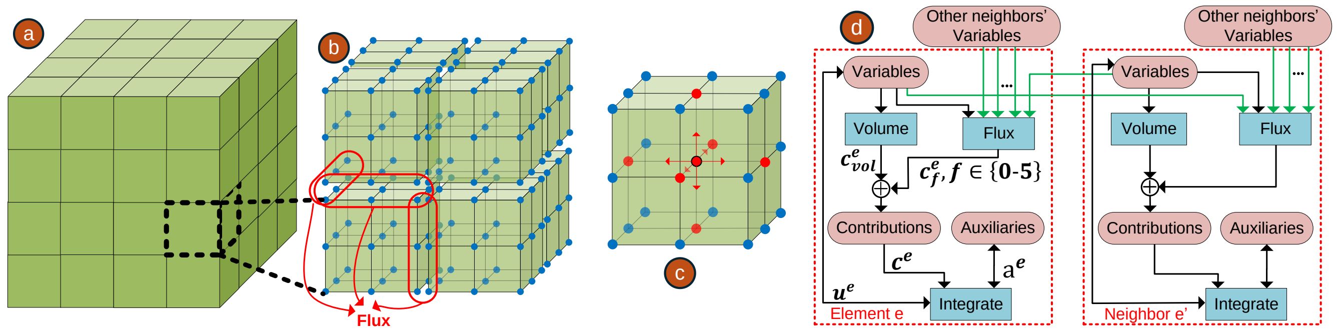

Fig. 1: (a) The 3D space is discretized into a structured mesh of straight-faced hexahedral elements that consist of $N \times N \times N$ nodes (b), where the unknown wave variables are computed. Flux computations resolve discontinuities of variables between neighboring elements. Using dG, Flux only requires information on the direct face of neighbors. Volume computations (c) are local and, when using GLL, require dot-products involving nodes along all three dimensions. Finally, (d) shows the dataflow graph of a wave simulation.

- We evaluate `HighWave` on elastic and acoustic wave equations using a Stratix 10 MX FPGA. When normalizing for equivalent peak HBM bandwidth, it outperforms state-of-the-art GPU implementations on an Nvidia V100 by up to 48%, with up to 2.84× higher energy efficiency.

II. Related Work

Wave simulations using dG are traditionally executed on HPC clusters, leveraging many-core CPU nodes to handle large-scale problems [13]. However, despite the high complexity of rewriting legacy scientific code for alternative architectures, recent GPU implementations have shown significant performance gains over CPUs [14]–[16]. To the best of our knowledge, `GAPS` [9] is the state-of-the-art GPU implementation targeting our setup of dG+GLL-based wave simulations. It outperforms CPU implementations by up to 84× on an Nvidia V100, achieving throughput up to $\sim$1 TFLOPs.

Several FPGA accelerators for dG-based wave simulations have been proposed. For example, the work in [17] accelerates a dG solver for Maxwell's equations, outperforming CPUs by 2×. The authors in [18]–[20] implement dG-based shallow-water models using triangular elements. These works target different applications on unstructured meshes and follow non-GLL solutions with high arithmetic intensity. None of these works leverage HBM. Within the scope of existing studies, `FAWS` [21] is the closest FPGA implementation to our work targeting dG-based wave simulations with GLL quadrature points on structured meshes, but it also does not use HBM and scales poorly with peak memory bandwidth.

HBM integration with FPGAs has shown significant performance gains in various applications, outperforming both GPUs and traditional FPGA designs without HBM. These applications span domains such as machine learning [22]–[25], quantum chemistry [26] and HPC [27]–[32]. While some works have employed HBM-equipped FPGAs for memory-bound PDE solvers, they focus on different methodologies. For example, NERO [33] accelerates compound stencil computations essential to weather prediction models, and FP-AMG [34] targets algebraic multigrid solvers. However, no work has leveraged HBM for dG-based solvers. This paper introduces the first scalable FPGA+HBM accelerator for dG-based large-scale wave simulations. Our architecture is highly efficient and general to allow adaptation over a wide range of hyperbolic PDE problems, including the acoustic and elastic wave equations demonstrated in this paper.

Algorithm 1: Runge-Kutta time stepping

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

III. Background

This section briefly outlines necessary background on dG-based wave simulations and the HBM architecture of Stratix.

A. Discontinuous Galerkin and Wave Simulations

Hyperbolic PDEs are typically solved through discretization in space and time. In this work, we discretize the 3D spatial domain using a structured mesh with straight-faced hexahedral elements (Fig. 1a), with $N \times N \times N$ nodes each (e.g. N=3 in Fig. 1b). Higher N means higher discretization order, which reduces dispersion errors in wave simulations. Additionally, a higher N is advantageous from a hardware perspective [9]. While our approach supports any N, we focus on N=8 in our evaluation (Sec. V), similar to prior works [9], [21] that focus on practical real-world applications. The goal is to compute the vector of 3D tensors $\mathbf{u}^e$ for each element e, which represents the PDE's unknown variables on the element's N^3 nodes.

Temporal discretization iteratively updates the solution vectors $\mathbf{u}^e$ of each element for the simulation period. For integration, we use a Runge-Kutta time-stepping scheme (Alg. 1), which updates $\mathbf{u}^e$ using an auxiliary vector $\mathbf{a}^e$. Two key computational kernels, *Volume* ($\mathcal{V}$) and *Flux* ($\mathcal{F}$), contribute to calculating $\mathbf{u}^e$ and $\mathbf{a}^e$ by producing the intermediate contribution vector $\mathbf{c}^e$. In our case, we focus on the Gauss-Lobatto-Legendre (GLL) quadrature integration method, which greatly reduces the amount of computations of *Volume* and *Flux*, but increases the communication-to-computation ratio [35].

Fig. 1d illustrates the dataflow graph of Alg. 1 when using dG. In each time step and for each element, the *Volume* kernel performs local computations that require only the element's own nodes. To calculate the $\mathbf{c}^e_{vol}$ contribution vectors, the

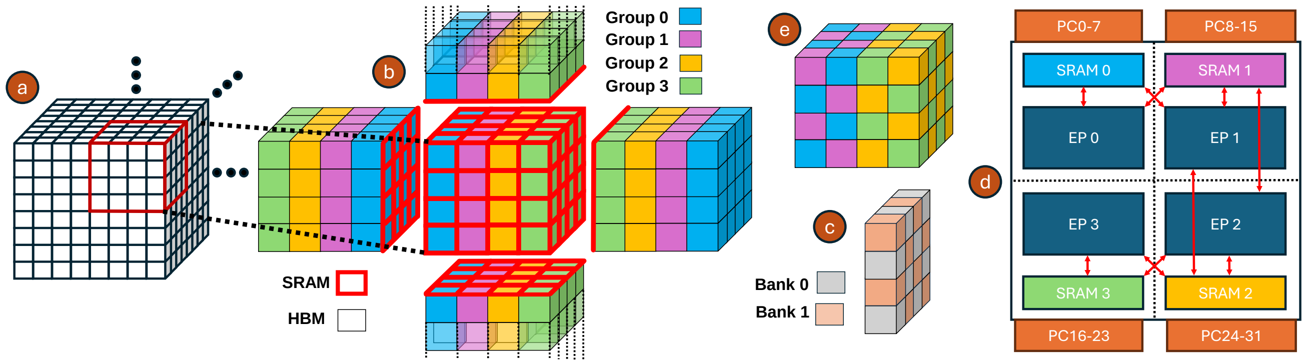

Fig. 2: Elements are packed into $4{\times}4{\times}4$ *blocks* (a). Red borders indicate elements needed on-chip to compute the center *block*. Each *block* is divided into 4 slices assigned to 4 processing groups (b), with each group partitioned to two banks (c) and assigned to a different set of HBM PCs and a distinct EP (d). The *balanced* grouping shown in (e) evenly distributes the left and right faces among all available PCs.

derivatives of each variable across all three spatial dimensions are computed (Fig. 1c) for each node. In the case of GLL, these derivatives are reduced to simple dot-product operations between a subset of variables and constant differentiation vectors. By contrast, *Flux* computations are non-local and resolve discontinuities between faces of neighboring elements. Unlike FDM solvers, which also require deeper neighbor data, in dG, *Flux* operations are limited to the face nodes of adjacent elements (Fig. 1b). For each element face, when using GLL, the *Flux* kernel performs lightweight scalar operations using variables on the N^2 face nodes of e and neighbor e', calculating the $\mathbf{c}_f^e$ contributions on these nodes. Finally, an *Integrate* kernel updates $\mathbf{u}^e$ and $\mathbf{a}^e$ in each step, using the contributions $\mathbf{c}^e$ and the element's mass inverse matrix $\mathbf{M^{-1}}$.

There are several ways of implementing this dataflow. As in prior works [9], [21], we employ kernel fusion to improve data locality and reduce traffic to main memory. With this approach, each element is fully processed before proceeding to the next. To do so, we load the element's data from main memory, along with the necessary neighbors' face variables, and then execute the three kernels for that element. Afterwards, we store the element's updated variables/auxiliaries in main memory and move to the next. Note that kernel fusion introduces a data hazard between variable updates of neighboring elements. Specifically, if *Flux* runs on a face shared with an already processed neighbor, it will read that neighbor's updated variables, causing a write-after-read hazard. To address this, *Flux* computes the contributions both for its own and each neighbor's face. We then use a flag mechanism to indicate that any neighbor that is processed afterwards does not need to re-run *Flux* for that face. Moreover, this dataflow allows multiple elements to be processed in parallel. However, processing two neighboring elements concurrently can introduce race conditions on the calculation of *Flux* contributions between shared face nodes. This can be resolved via synchronization of faces or scheduling of elements in a way that guarantees no neighbors are processed in parallel.

B. High Bandwidth Memory on the Stratix 10 MX

HBM is a 3D-stacked DRAM technology. Specifically, in the HBM2 standard, each HBM stack consists of a number of dies, with two independent memory channels per die. Each channel is further split into two pseudochannels (PCs), each with their own controller and address space, but shared command buses. The Stratix 10 MX FPGA has two HBM2 stacks, one at the top and one at the bottom of the device, with four dies per stack, for a total of 16 channels or 32 PCs. FPGA logic communicates with the memory controller of each PC via 256-bit AXI or Avalon interfaces. While the maximum theoretical frequency of these interfaces is 500 MHz, Intel recommends limiting it to 405 MHz [36], dropping the theoretical peak bandwidth from 512 GB/s to 414.72 GB/s.

IV. HighWave System Design

In this section, we first propose memory layout and mapping techniques that maximize memory performance and then we delineate our system architecture. We showcase our approach for the elastic equation, which involves six stress and three velocity variables, amounting to nine variables. Consequently, the elastic wave solver uses nine contributions, nine auxiliaries and one mass inverse value per element node. Collectively, we refer to a node's distinct values as quantities, yielding a total of 28 ($=3{\cdot}9{+}1$) quantities per node. Our approach can be applied in straightforward fashion to other wave equations, such as acoustic (Sec.V), electromagnetic etc.

A. Memory Layout and Mapping

`HighWave` assumes that element data is stored in DRAM. As with other accelerators, it fetches data from HBM DRAM into local on-chip memory and then operates on this data via a custom array of Element Processors (EPs) that execute out of local memory. Fig. 2 shows an overview of the mapping of data onto local memories and HBM channels.

Element batching: Prior FPGA accelerators [21] for wave simulations scale poorly with the number of memory channels, as they fetch and process single elements at a time from DRAM. As the number of channels increases, this reduces the consecutive transaction size to each channel, degrading bandwidth. To address this, we propose fetching and processing a large, contiguous 3D *block* of elements at a time (Fig. 2(a)), maintaining large, consecutive per-channel transactions. This enables scalability, while improving locality by reducing off-chip accesses for neighbor data, as most neighbors of elements within a *block* are stored on-chip. Note that processing of elements at the border of a *block* also requires accesses to data of external neighbors. We will call such accesses *irregular*, in contrast to *regular* accesses for fetching the *block* itself.

Each *block* can then be treated as a larger element that requires special computation and follows the dataflow from Sec. III-A. Fig. 2(b) shows all the element data that gets fetched on-chip (marked in red borders) to process each *block*, comprising all elements in the *block* and faces of elements on neighboring *blocks*. As mentioned, when processing a *block*, *Flux* updates the contributions on each neighboring *block's* face, where a flag is used to indicate that a shared face has already been processed. As such, the number of neighboring *block* faces fetched from HBM ranges from zero to six, with an average of three, when using hexahedral elements.

For a given *block* size, the shape that minimizes *irregular* accesses and maximizes locality is cubic ($A \times A \times A$). An $A \times A \times A$ cube reduces *irregular* accesses by a factor of A. For example, in a 4×4×4 *block*, *irregular* transactions are proportional to the number of external neighboring faces, which totals 6×16=96 (6 faces per *block*, 16 element faces per *block* face). In contrast, loading and processing elements individually requires 6×64=384 neighboring faces (6 faces per element, 64 elements in the *block*). This equals a 4× reduction in *irregular* transfers. Without loss of generality, for the rest of this paper, we will use 4×4×4 *blocks*, the largest size that fits in on-chip memory for our FPGA when $N{=}8$ in the elastic wave. Nonetheless, the proposed method is adaptable to different *block* dimensions (e.g. 2×4×4, 8×4×4 etc.), based on the value of N and on-chip memory capacity.

Data mapping to EPs and HBM channels: To achieve load balancing of a *block's* data across HBM channels and parallel EPs, we divide the 3D *block* into four groups, as illustrated in Fig. 2b. Each group, represented by a distinct color, corresponds to a specific 1×4×4 slice of elements along the x-axis within a *block*, mapped to a different set of HBM channels and processed by a different EP. EPs are physically placed near their respective channels (Fig. 2(d)). During processing, elements within a slice exchange face variables and contributions with: (i) other elements in the same slice, (ii) elements in adjacent slices within the *block*, and (iii) elements in neighboring *blocks* (external neighbors). To ensure that each slice's external neighbors belong to the same set of PCs, the group order within each *block* is flipped in each subsequent *block* along the x-axis (Fig. 2(b)). This flipping simplifies control logic and does not require long data paths to move external neighbor data between EPs and distant PCs.

Each slice contains 16 elements and is mapped to a set of four HBM channels, enabling utilization of all 16 channels across four groups. The 28 element quantities of each slice are divided into four sets of seven quantities each, with each set mapped to a different channel. Since channels consist of two PCs, each quantity set is further partitioned into two banks, where each bank holds data of 8 out of the 16 elements and is assigned to a different PC of the corresponding channel (Fig. 2(c)). This enables a large, consecutive access (128kB in our case, when $N{=}8$) for each PC during *regular* transactions,

TABLE I: Mapping of group 0's quantities to all eight PCs.

PC	Quantities	PC	Quantities
0	mass_inv, var0–5 (Bank 0)	**4**	contr4–8, aux0–1 (Bank 0)
1	mass_inv, var0–5 (Bank 1)	**5**	contr4–8, aux0–1 (Bank 1)
2	var6–8, contr0–3 (Bank 0)	**6**	aux2–8 (Bank 0)
3	var6–8, contr0–3 (Bank 1)	**7**	aux2–8 (Bank 1)

which leads to very efficient bandwidth utilization. Bandwidth utilization is lower when fetching neighboring *block* faces, due to the high *irregularity* in their access patterns. Table I shows an example mapping of quantities to PCs for Group 0, with other groups following similar patterns. This mapping uses all 32 PCs available on the MX device. The 256-bit interface of each PC can fit eight 32-bit floating-point quantities per word, resulting in one zero-padded 32-bit value per 256-bit word when assigning seven quantities per PC.

The grouping of elements into slices shown in Fig. 2(b) does not evenly distribute the external neighbors of a *block's* left and right faces across all PCs, causing only eight PCs to be used when fetching these faces, which degrades performance. To address this, we propose two optimizations. First, we schedule *blocks* to traverse each row along the x-axis with a stride of 2, accessing even-indexed *blocks* first, then odd-indexed ones This ensures that either both or none of a *block's* left/right faces need to be fetched. When fetched, they are handled concurrently, using 16 PCs in total. Second, the element grouping can be modified, as shown in Fig. 2(e), to evenly split the left and right face between groups 0-1 and 2-3, respectively. This grouping, referred to as *balanced*, fully utilizes all PCs for all *block* faces, improving performance by up to 3% (Sec. V). However, it requires a slightly more complex design and scheduling to preserve data dependencies, not illustrated here for brevity.

B. System Architecture

Fig. 3 shows our proposed spatial architecture, comprising a configurable number of EPs and custom, dedicated load/store (LS) units that move each group's data between HBM PCs and the corresponding EP's on-chip buffers. The number of EPs is determined by the number of PCs (and, thus, peak bandwidth) and available on-chip resources. Before execution begins, initial data is offloaded to HBM from the host via the PCIe bus. Upon completion, the host reads back the results.

EP kernels: The EP microarchitecture, Fig. 3 (right), is designed to saturate HBM bandwidth with minimum resource overhead. It consists of three compute kernels, *Volume*, *Flux* and *Integrate*, utilizing kernel fusion. The *Volume* is the most computationally intensive kernel. It calculates the $\mathbf{c}^{e}_{vol}$ vector with the volume contributions of each element's nodes. To do so, it iterates over each node and computes dot products between a subset of variable vectors and constant differentiation vectors of length N across all three dimensions. Parallel multipliers and adder trees are used for the dot-products. We implement the *Volume* loop with an initiation interval (II) of 1, thus requiring one clock cycle per node for a total of N^3 cycles per element. To enable an II of 1, a large number of memory accesses are needed to each variable vector per clock

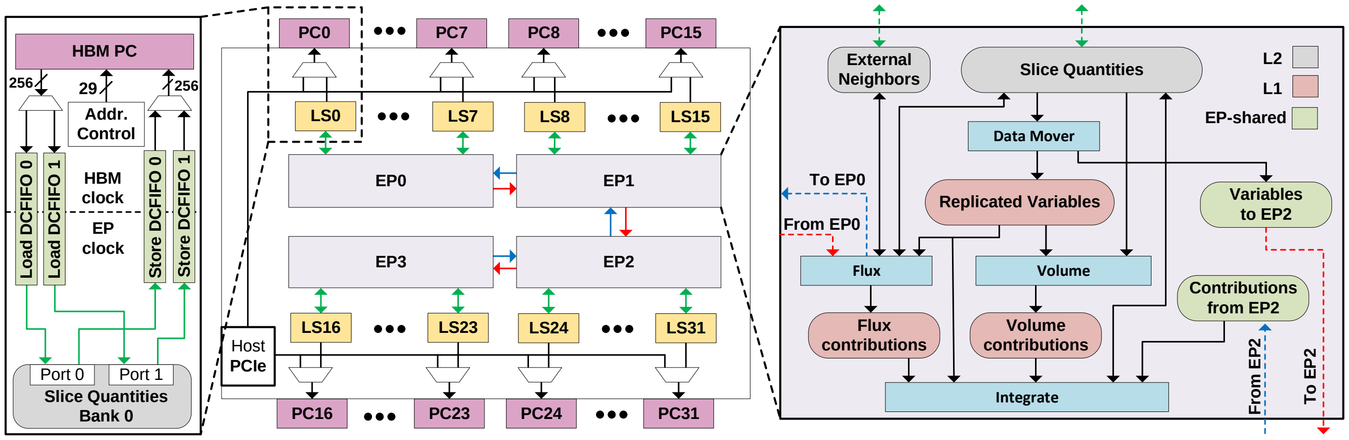

Fig. 3: System Architecture (middle), including the internal architecture of LS units (left) and EPs (right).

cycle (up to 22 when $N{=}8$) to feed all the dot-product engines. This requires replication of variables in on-chip buffers.

The *Flux* kernel calculates the 2D $\mathbf{c}_f^e$ and $\mathbf{c}_f^{e'}$ flux contribution vectors on the shared face f of elements e and e'. For each face, it iterates over all N^2 nodes and performs vector addition and scaling as a function of the variables on the face nodes of both elements. This loop is also pipelined with an II of 1. For a given element, this process is repeated for each face not already processed by a neighbor. The maximum number of cycles needed to compute *Flux* for one element is $6{\times}N^2$. Finally, the *Integrate* kernel consists of a single pipelined loop (II=1) that iterates over all N^3 nodes of an element and updates the variable and auxiliary tensors $\mathbf{u}^e$ and $\mathbf{a}^e$ based on the *Volume* and *Flux* contributions, using scalar operations.

Memory architecture: Internally, EPs incorporate two memory hierarchy levels, L1 and L2, as in Fig. 3 (right). A *Data Mover* operating in parallel with the compute kernels manages data movement from L2 to L1. L2 is connected to the LS units and stores quantities of all 16 elements in a group, as well as face data of the group's external neighbors. For the group's internal elements, we use one buffer per quantity, holding $16{\cdot}N^3$ values and partitioned to two banks (Sec. IV-A). For the external neighbor data, we need one buffer for each variable and contribution. In the *balanced* grouping, each group has 24 external neighboring faces, so each of these buffers stores $24{\cdot}N^2$ values. All these memories are double-buffered to overlap communication and computation.

As mentioned above, the *Volume* kernel performs simultaneous memory accesses to the variable tensor (up to 22 in elastic when $N{=}8$), requiring significant replication of the variables to feed all of its dot-product engines. However, replicating the entire slice's variables at this scale is infeasible due to memory constraints. To resolve this, we use L1 memory to act as a cache holding the replicated variables of the current element that the EP processes. L1 is also double-buffered to overlap computation with prefetching of the next element's variables from L2. Each EP also includes small buffers in L1 to hold *Volume* and *Flux* contributions that will be used by *Integrate*.

Intra-EP communication: Next, we outline the connections between an EP's compute kernels and its buffers (Fig. 3 right). *Volume* reads from L1 and writes to the *Volume* contributions. It also reads contributions from L2 to accumulate any contributions on face nodes already calculated by external neighbors. *Flux* reads the variables of the external neighbors and updates their contributions in L2. It does the same for neighbors inside the EP's slice, hence its *Slice Quantities* connection. It also reads the variables of the current element directly from L1. *Integrate* reads the current element's variables from L1, as well as calculated *Volume*/*Flux* contributions. It writes updated auxiliaries and variables back to L2.

Inter-EP communication: EPs that process neighboring slices need to exchange variables and contributions on their shared face. Since EPs are placed on distinct regions of the programmable fabric, we aim to minimize the wire length required for cross-EP communication. This will alleviate congestion on the routing fabric and maximize operating frequency. To do so, we instantiate only one additional copy of the variables in L1, placed at the border of adjacent EPs. A similar dedicated buffer is used for exchange of contributions. These two buffers, labeled as *EP-shared*, facilitate inter-EP communication. During *Flux*, using the mapping in Fig. 2(b), we use a convention that faces shared by adjacent EPs are processed by the EP with the higher index. For example, to process the face shared between slices belonging to groups 1 and 2, EP2 reads the face variables and writes back the calculated contributions from/to EP1's *EP-shared* buffer. EP1 then reads these contributions during its *Integrate* stage. The same process occurs between EP0–EP1, and EP2–EP3, as indicated by blue and red arrows in Fig. 3. The *balanced* grouping in Fig. 2(e) requires one additional set of *EP-shared* buffers for EP0–EP3 communication, not shown for simplicity.

Clock domains and LS units: To maximize bandwidth, HBM controllers must run at maximum frequency (405 MHz in Stratix). However, EPs may not need such a high frequency to saturate the bandwidth. In such cases, using the same clock domain for the HBM controllers and the EPs will lead to worse energy efficiency and make timing closure more challenging. As a result, our architecture operates on two different clock domains, one for the EPs and one for the HBM controllers.

The LS units are responsible for moving data between HBM PCs and the EP L2 buffers. Fig. 3 (left) shows the LS architecture, consisting of control and address generation logic, as

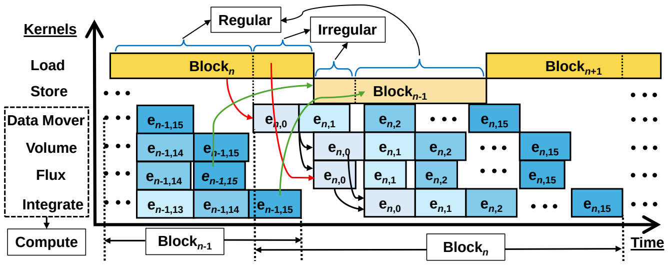

Fig. 4: System Scheduling.

well as clock-crossing FIFOs to move data between the two clock domains. LS units dynamically decide which neighbors need to be fetched and follow a static scheduling of *blocks* that eliminates race conditions. For clock crossing, we can use one or two dual-clock FIFOs (DCFIFOs) in each direction. Using a single DCFIFO can degrade Store performance if the EP clock is slower. It also requires a large enough Load FIFO depth to maintain bandwidth during Load. Depending on the FIFO implementation, such high depth can be expensive resource-wise. Conversely, using two DCFIFOs per direction (Fig. 3) and alternating between them from the HBM side doubles the data rate from the EP side, eliminating Load/Store bandwidth degradation under minimum FIFO depth requirements.

C. System Scheduling

Fig. 4 illustrates the system's execution schedule. As mentioned, L2s are double-buffered, allowing processing to be overlapped with load/store of next/previous *blocks*. We show kernels of one EP, but all EPs are launched simultaneously and operate in parallel. LS operations are divided into *regular* and *irregular* categories: *regular* operations handle fetching and storing data for the current *block* being processed, while *irregular* operations manage data for neighboring *blocks*. Only one of these four operations can access the HBM PCs at a time, ensuring that memory bandwidth is used without contention.

Connections marked by red and green arrows indicate dependencies between load-compute and compute-store, respectively. Dependencies between the EP's internal kernels are marked in black. The *Data Mover* begins fetching element 0 from L2 to L1 once the *regular* load completes. The *Volume* and *Flux* kernels are scheduled to start in parallel once the *Data Mover* has transferred element 0. Since *Flux* requires neighbor data, it also waits until *irregular* load operations are completed. *Integrate* begins after element 0 has been processed by both *Volume* and *Flux*. Lastly, the *irregular* store can start when *Flux* finishes the final element, and the *regular* store after *Integrate* has processed the last element in a *block*. Within each slice, elements are processed in an order that guarantees no memory bank conflicts between memory accesses of different kernels. This eliminates arbitration when accessing the L2 and L1 buffers, thus optimizing the system's interconnects. EPs of neighboring slices follow the same element order to guarantee correct exchange of data through the EP-shared buffers.

V. Evaluation

We evaluated `HighWave` on an Intel Stratix 10 MX 2100, targeting elastic and acoustic wave simulations on proxy problems representative of real-world applications, such as full-wavefield inversion in hydrocarbon exploration. $N=8$ and FP32 precision has been used throughout. We used Intel High-Level-Synthesis (HLS) to design the *Volume*, *Flux* and *Integrate* kernels, while the rest of the system (LS units, memory architecture, control logic etc.) is implemented in Verilog. For elastic, we use 4 EPs operating on a $4\times4\times4$ *block* (Sec. IV). The acoustic equation involves four unknowns (13 quantities total). This leads to less memory size per element, supporting 8 EPs operating on an $8\times4\times4$ *block*.

We compare against `GAPS` [9] running on an Nvidia P100 and V100 GPU. All devices incorporate 16GB of HBM2 and their key specifications are shown in Table II. Measurements correspond to $32\times32\times32$ grids (32,768 elements, $\sim$1.75 GiB total size), the largest representative problem size that fits in HBM, with similar results obtained for smaller sizes. Both the GPU and FPGA run fully independently once launched. Results exclude overheads for mesh generation and initial/final data transfer between the host CPU and the device's HBM, as they were less than 0.5% of total runtime.

TABLE II: Hardware platform specifications.

	Stratix 10 MX 2100	P100	V100
On-chip memory	16.73 MB	22.81 MB	35.56 MB
Peak FLOPs	6.3 TFLOPs	9.3 TFLOPs	14.1 TFLOPs
Peak HBM BW	414.7 GB/s	734 GB/s	900 GB/s
Technology node	Intel 14nm	TSMC 16nm	TSMC 12nm

TABLE III: Mapping of group quantities using 7 out of 8 PCs.

PC	Quantities	PC	Quantities
0	mass_inv, var0–6 (Bank 0)	**4**	contr6–8, aux0–4 (Bank 0)
1	mass_inv, var0–6 (Bank 1)	**5**	contr6–8, aux0–4 (Bank 1)
2	var7–8, contr0–5 (Bank 0)	**6**	aux5–8 (Banks 0, 1)
3	var7–8, contr0–5 (Bank 1)	**7**	Unused

A. Implementation Results

We describe several important, device-specific implementation choices needed to maximize performance on Stratix. First, as also identified in [22], timing closure is particularly challenging near the device's bottom-left PCs (PC16 and PC17), due to their proximity to the secure device manager. To address this in the elastic case, instead of zero-padding each PC of EP3, we exclude PC16 and re-distribute its quantities to fill the zero-padded slots of the other seven PCs (Table III). We do the same for all EPs, maintaining the performance of the original configuration, while enabling timing closure and reducing HBM power usage, as four of the 32 PCs are inactive. We apply similar optimizations for acoustic, using 26 PCs in total, but this requires some PCs to contain quantities of two neighboring groups. Overall, $32 \mod K$ PCs are unused with this scheme, where K is the number of quantities. To further reduce routing congestion, we connect the PCIe interface only to channels that require initialization or read-back.

Second, choosing how the LS DCFIFOs are implemented is a key design decision. On Stratix, they can be implemented using either M20K blocks or MLABs (soft logic) [37]. The first choice requires at least seven M20Ks per 256-bit FIFO, which offer a FIFO depth of 512 words [38]. Depending on the EP frequency, this depth may not suffice and lead to performance

TABLE IV: Resource usage breakdown of one elastic EP.

Components	DSPs	M20Ks	ALMs
Volume	172 (4.3%)	1 (0%)	4300 (0.6%)
Flux	61 (1.5%)	3 (0%)	3570 (0.5%)
Integrate	37 (0.9%)	1 (0%)	1760 (0.2%)
Data Mover	0 (0%)	0 (0%)	40 (0%)
L2	0 (0%)	1040 (15.2%)	0 (0%)
L1	0 (0%)	360 (5.3%)	0 (0%)
EP-shared	0 (0%)	36 (0.5%)	0 (0%)
IC/Pipelining	0 (0%)	0 (0%)	22718 (3.2%)
Total	**270 (6.8%)**	**1441 (21.0%)**	**32348 (4.6%)**

TABLE V: Total resource usage of `HighWave`.

Components	DSPs	M20Ks	ALMs
1 EP (elastic)	270 (6.8%)	1441 (21.0%)	32348 (4.6%)
1 EP (acoustic)	97 (2.4%)	620 (9.1%)	12618 (1.8%)
1 LS Unit (average)	0 (0%)	0.4 (0%)	1033 (0.1%)
Other (PCIe, HBM, IC)	0 (0%)	175 (2.6%)	71466 (10.2%)
Total (4EPs, Elastic)	**1080 (27.3%)**	**5951 (86.8%)**	**229775 (32.7%)**
Total (8EPs, Acoustic)	**776 (19.6%)**	**5153 (75.3%)**	**211641 (30.1%)**

degradation when using a single FIFO for Load (Sec. IV-B). As a result, Load may need a higher depth, or to use two DCFIFOs as in Fig. 3, doubling M20K usage in both cases. Simultaneously, Store requires 2 FIFOs to maintain bandwidth. This adds up to 896 M20Ks (13.1% of total) for the LS units of all PCs. However, the M20K blocks on the MX device are limited and to exploit locality, `HighWave` prioritizes use of on-chip SRAM for *block* buffering. As such, and since the ALM requirements of `HighWave` are much lower, we chose to use two DCFIFOs per direction implemented using MLABs. The minimum FIFO depth requirements of this scheme led to only 4.1% total ALM utilization for all LS units.

Table IV shows the resource breakdown of an elastic wave EP, including the compute kernels, the *Data Mover*, the L1, L2 and *EP-shared* buffers, as well as resources for interconnect (IC) logic and pipeline stages on paths to L1/L2 buffers. Overall, M20Ks are the most critical resource, while ALM and DSP utilization is small, due to the low arithmetic intensity. Table V also shows the resources of an acoustic EP, and for the full system, including LS units and system-level components (e.g. PCIe, HBM controllers and interconnects/pipelines on data paths to HBM channels). The acoustic wave has even lower arithmetic intensity, requiring fewer resources.

Fig. 5 shows the floorplan for the elastic wave implementation, indicating the location of the HBM PCs and the EPs, as well as the routing heatmap. Overall, congestion was highest along paths between PCs and L2 buffers. Using small *EP-shared* buffers on the border between neighboring EPs greatly reduced routing congestion and contributed to timing closure.

Synthesized EPs were able to reach a maximum frequency of 370 MHz. Using deep pipelining to the PCs, we managed to meet timing at 400 MHz for the HBM controllers, which nears the 405 MHz maximum that Intel recommends.

B. Performance Analysis

Memory performance: Our design supports a peak theoretical bandwidth of 409.6 GB/s when utilizing all 32 PCs. To determine the maximum attainable memory bandwidth, we tested the system running the elastic wave with only the LS units active, isolating memory performance from potential

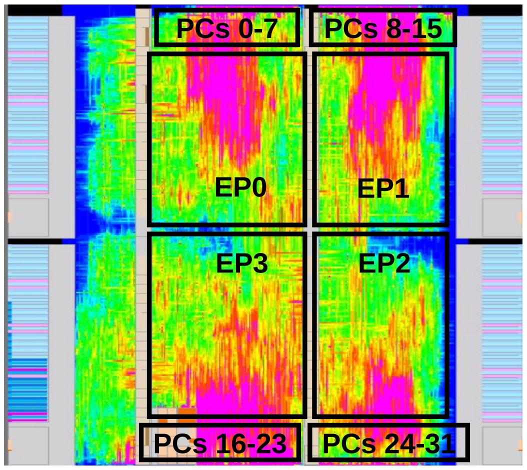

Fig. 5: Architecture floorplan with routing heatmap.

computation stalls. *Regular* HBM accesses, characterized by large consecutive transactions, achieved ∼90% per-channel utilization, which translates to 78% of peak BW when excluding 4 PCs. Next, introducing *irregular* transactions drops effective bandwidth to 287.3 GB/s (70.1% of peak). The *balanced* element grouping (Sec. IV-A), which evenly distributes left and right face data across all PCs, increased bandwidth to 295.6 GB/s (72.2% of peak), yielding a modest 3% speedup. This limited improvement reflects the small size of left and right neighbor data compared to the total data, compounded by a 4× reduction in *irregular* transactions. For acoustic, bandwidth was lower at 66.2%, due to using two less PCs. Prior works for the same elastic problem setup achieved significantly less utilization. `GAPS` [9] reached only about 56% efficiency on an Nvidia V100 GPU, while the FPGA-based approach in [21] achieved 55% on single-channel DDR4 and dropped further to 34.5% when scaled to four DDR4 channels.

System performance: As communication and computation are overlapped, we need just enough compute power to fully saturate the attainable memory bandwidth. A faster computation is wasteful in power consumption, while a slower computation introduces stalls that degrade performance. For a given number of EPs, we can calculate the clock cycles needed to process a *block*, when using the schedule from Sec. IV-C. The EP clock frequency can then be set to match the communication speed. However, the load/store time of a *block* varies based on how many neighboring *blocks* need to be read from and written to HBM, ranging between zero and six. Any time computation is slower, bubbles are added. The shortest communication time appears in cases when no neighbors need to be fetched, which is very rare. Overall, a 355 MHz EP frequency was enough to eliminate most stalls and saturate HBM bandwidth for both applications.

Fig. 6 (left) shows the performance of the elastic system at different EP operating frequencies for the two system variants (original *vs. balanced* element grouping). As expected, the *balanced* grouping leads to better bandwidth utilization and, thus, throughput. We observe a relatively small throughput decrease (6.4%) when frequency drops from 360 MHz to 320 MHz (11.1% frequency drop), showcasing a weak dependence of

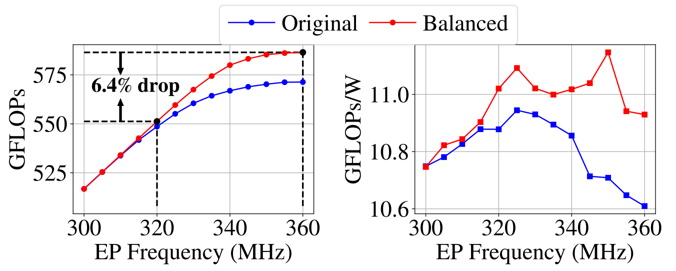

Fig. 6: Elastic wave performance (left) and energy efficiency (right) *vs.* EP frequency of original and *balanced* element grouping.

performance on a wide frequency range. This is because the aforementioned compute stalls only appear in cases where a few neighboring *blocks* need to be fetched. Communication of more neighbors leads to fewer bubbles as computation is faster than communication. Performance degrades faster for frequencies below 320 MHz, due to the addition of bubbles in more cases. Both variants achieve the same throughput at low frequencies, as they are both compute-bound.

Fig. 7 shows the roofline plot of `HighWave` and `GAPS`, denoting the low arithmetic intensity of the two applications. Our 4× reduction of *irregular* HBM accesses slightly increases the arithmetic intensity over `GAPS`. Fig. 8 (left) illustrates a throughput comparison against `GAPS` running on an Nvidia P100 and V100. The acoustic wave exhibits lower arithmetic intensity than the elastic wave, hence its lower overall GFLOPs. The P100 is constrained by small L1 cache size (7.7× smaller than V100), thus its much smaller performance than V100 and Stratix. The V100 does not suffer from this bottleneck and it achieves higher throughput than Stratix, due to its much higher peak bandwidth (Table II). As the bottleneck in all cases is attainable HBM bandwidth, we enable a fair comparison by normalizing V100 throughput to match the peak bandwidth of Stratix (409.6 GB/s). When doing so, `HighWave` is 36% and 48% faster than V100 for elastic and acoustic, respectively. This showcases better bandwidth utilization, attributed to our proposed memory optimizations that improve locality and access patterns, while consistently issuing sufficiently large transfer bursts to all PCs.

C. Power Analysis

To measure power for Stratix, we used the Intel Board Test System's power monitor under standard room temperature. We configure `HighWave` to run elastic and acoustic waves for tens of thousands of time steps, recording average power over several minutes. Total power consumption for the elastic wave ranged from 48.1–53.9W depending on EP frequency. Up to 14.8W (27% of total) was consumed by HBM. Maximum power for the acoustic wave was 46W. Fig. 6 (right) shows the energy efficiency (in GFLOPs/Watt) of the elastic system at different EP frequencies. The *balanced* grouping at 350 MHz yielded maximum energy efficiency. However, efficiency varied by less than 2% across the frequency range. Similar trends were observed for the acoustic wave.

To measure GPU power, we used nvidia-smi. Fig. 8 (right) compares the energy efficiency of `HighWave` with P100 and V100. Similarly to throughput, we observe lower energy

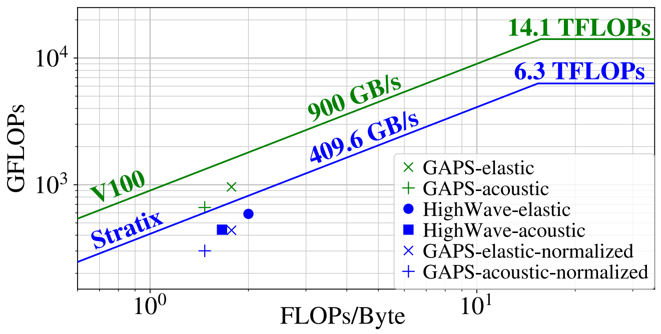

Fig. 7: Roofline models for Stratix 10 MX and Nvidia V100.

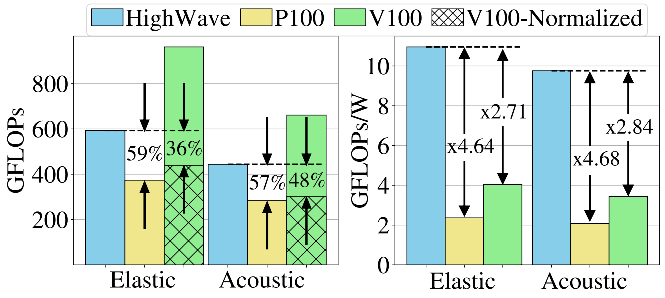

Fig. 8: Throughput (left) and energy efficiency (right) comparison of `HighWave` *vs.* `GAPS` running on the P100 and the V100.

efficiency for the acoustic wave, due to its lower arithmetic intensity, which increases the ratio of energy spent for communication rather than computation. Overall, `HighWave` on Stratix is 4.64× and 2.71× more energy efficient than P100 and V100, respectively, for the elastic wave. For the acoustic wave, it is 4.68× and 2.84× more energy efficient. This is attributed to our custom, low-area LS units, which provide high memory performance, complemented by efficient EPs that saturate bandwidth under minimum resource requirements.

VI. Summary and Conclusions

In this work, we presented `HighWave`, the first HBM-enabled FPGA accelerator targeting memory-bound, dG-based wave simulations. Our design features a scalable spatial architecture composed of highly efficient EPs and employs a wide range of optimizations to maximize data reuse and HBM bandwidth. We evaluate `HighWave` on the acoustic and elastic wave equations, achieving up to 72.2% bandwidth utilization, despite the high irregularity in access patterns. When normalizing for equivalent HBM bandwidth, `HighWave` is 36% and 48% faster than an Nvidia V100 on elastic and acoustic, respectively, with 2.71× and 2.84× higher energy efficiency. These results highlight the potential of HBM-enabled FPGAs, when optimized for dataflow and memory efficiency, to rival GPUs in highly memory-bound workloads, a domain where GPUs traditionally excel. In future work, we plan to extend `HighWave` to multi-FPGA solutions for larger problem sizes and develop a design generator to automatically configure and instantiate `HighWave` for diverse applications and platforms.

Acknowledgment

This research was supported by ExxonMobil Technology and Engineering Company.

REFERENCES

[1] A. Fathi, L. F. Kallivokas, and B. Poursartip, "Full-waveform inversion in three-dimensional PML-truncated elastic media," *Computer Methods in Applied Mechanics and Engineering*, vol. 296, pp. 39–72, 2015.

[2] D. Lumley, "4D seismic monitoring of CO2 sequestration," *The Leading Edge*, vol. 29, no. 2, pp. 150–155, 2010.

[3] 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.

[4] H. Wang, I. Sihar, R. Pagán Muñoz, and M. Hornikx, "Room acoustics modelling in the time-domain with the nodal discontinuous Galerkin method," *The Journal of the Acoustical Society of America*, vol. 145, no. 4, pp. 2650–2663, 2019.

[5] L. Maio and P. Fromme, "On ultrasound propagation in composite laminates: Advances in numerical simulation," *Progress in Aerospace Sciences*, vol. 129, p. 100791, 2022.

[6] 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.

[7] K.-H. Barth, "The Politics of Seismology: Nuclear Testing, Arms Control, and the Transformation of a Discipline," *Social Studies of Science*, vol. 33, no. 5, pp. 743–781, 2003.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] R. Gandham, D. Medina, and T. Warburton, "GPU accelerated discontinuous Galerkin methods for shallow water equations," *Communications in Computational Physics*, vol. 18, no. 1, pp. 37–64, 2015.

[15] 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.

[16] D. S. Abdi, L. C. Wilcox, T. C. Warburton, and F. X. Giraldo, "A GPU-accelerated continuous and discontinuous Galerkin non-hydrostatic atmospheric model," *The International Journal of High Performance Computing Applications*, vol. 33, no. 1, pp. 81–109, 2019.

[17] 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.

[18] 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.

[19] M. Büttner, C. Alt, T. Kenter, H. Köstler, C. Plessl, and V. Aizinger, "Enabling Performance Portability for Shallow Water Equations on CPUs, GPUs, and FPGAs with SYCL," in *Platform for Advanced Scientific Computing Conference (PASC)*, 2024.

[20] J. Faj, T. Kenter, S. Faghih-Naini, C. Plessl, and V. Aizinger, "Scalable multi-FPGA design of a discontinuous Galerkin shallow-water model on unstructured meshes," in *Platform for Advanced Scientific Computing Conference (PASC)*, 2023.

[21] D. Gourounas, B. Hanindhito, A. Fathi, D. Trenev, L. K. John, and A. Gerstlauer, "FAWS: FPGA Acceleration of Large-Scale Wave Simulations," in *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2023.

[22] M. Doumet, M. Stan, M. Hall, and V. Betz, "H2PIPE: High Throughput CNN Inference on FPGAs with High-Bandwidth Memory," in *International Conference on Field-Programmable Logic and Applications (FPL)*, 2024.

[23] T. Zhang, A. Boutros, S. Gribok, K. Boateng, and V. Betz, "A Software-Programmable Neural Processing Unit for Graph Neural Network Inference on FPGAs," in *International Conference on Field-Programmable Logic and Applications (FPL)*, 2024.

[24] S. K. Venkataramanaiah, H.-S. Suh, S. Yin, E. Nurvitadhi, A. Dasu, Y. Cao, and J.-s. Seo, "FPGA-based low-batch training accelerator for modern CNNs featuring high bandwidth memory," in *International Conference on Computer-Aided Design (ICCAD)*, 2020.

[25] S. Zeng, J. Liu, G. Dai, X. Yang, T. Fu, H. Wang, W. Ma, H. Sun, S. Li, Z. Huang *et al.*, "FlightLLM: Efficient Large Language Model Inference with a Complete Mapping Flow on FPGAs," in *International Symposium on Field Programmable Gate Arrays (ISFPGA)*, 2024.

[26] P. Stachura, G. Li, X. Wu, C. Plessl, and Z. Fang, "SERI: High-Throughput Streaming Acceleration of Electron Repulsion Integral Computation in Quantum Chemistry using HBM-based FPGAs," in *International Conference on Field-Programmable Logic and Applications (FPL)*, 2024.

[27] C. Liu, H. Liu, L. Zheng, Y. Huang, X. Ye, X. Liao, and H. Jin, "FNNG: A High-Performance FPGA-based Accelerator for K-Nearest Neighbor Graph Construction," in *International Symposium on Field Programmable Gate Arrays (ISFPGA)*, 2023.

[28] W. Jaiyeoba, N. Elyasi, C. Choi, and K. Skadron, "ACTS: A Near-Memory FPGA Graph Processing Framework," in *International Symposium on Field Programmable Gate Arrays (ISFPGA)*, 2023.

[29] Z. He, L. Song, R. F. Lucas, and J. Cong, "LevelST: Stream-based Accelerator for Sparse Triangular Solver," in *International Symposium on Field Programmable Gate Arrays (ISFPGA)*, 2024.

[30] C. Su, L. Du, T. Liang, Z. Lin, M. Wang, S. Sinha, and W. Zhang, "GraFlex: Flexible Graph Processing on FPGAs through Customized Scalable Interconnection Network," in *International Symposium on Field Programmable Gate Arrays (ISFPGA)*, 2024.

[31] X. Chen, F. Cheng, H. Tan, Y. Chen, B. He, W.-F. Wong, and D. Chen, "ThunderGP: Resource-efficient graph processing framework on FPGAs with HLS," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 4, pp. 1–31, 2022.

[32] K. Kamalakkannan, G. R. Mudalige, I. Z. Reguly, and S. A. Fahmy, "High-Level FPGA Accelerator Design for Structured-Mesh-Based Explicit Numerical Solvers," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2021.

[33] G. Singh, D. Diamantopoulos, C. Hagleitner, J. Gomez-Luna, S. Stuijk, O. Mutlu, and H. Corporaal, "NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling," in *International Conference on Field-Programmable Logic and Applications (FPL)*, 2020.

[34] P. Haghi, T. Geng, A. Guo, T. Wang, and M. Herbordt, "FP-AMG: FPGA-Based Acceleration Framework for Algebraic Multigrid Solvers," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020.

[35] 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, no. 3, pp. 165–198, 2015.

[36] Intel Corporation, "High Bandwidth Memory (HBM2) Interface FPGA IP User Guide," 2024.

[37] ——, "FIFO Intel® FPGA IP User Guide," 2024.

[38] ——, "Intel Stratix 10 Embedded Memory User Guide," 2023.