Optimizing an FPGA-based 3D FDTD Accelerator through HLS

Tan Jin Hung

January 14, 2026

Abstract

Contents

1 Introduction

The current technological era has enabled the use of computers to simulate our complex understanding of the physical world. However, simulations that accurately mimic real-life phenomena remain computationally intensive for many practical applications. To manage this complexity, these simulations are often decomposed down into their fundamental physical components. Among these fundamentals, the simulation of electromagnetic waves remains at the forefront of modern research, with the FDTD method serving as a primary tool for high-fidelity analysis in complex environments [1].

A primary method for simulating electromagnetic waves is the Finite-Difference Time-Domain (FDTD) method [2]. While FDTD is a cornerstone of computational electromagnetics, providing a robust foundation for analysis, it faces significant challenges in 3D space. Specifically, the iterative nature of the method creates bottlenecks in memory bandwidth and data throughput, often characterized as the "memory wall" in high-performance computing [3, 4]. As noted by Kong and Su [5], the computational demand of the 3D FDTD algorithm increases cubically with the grid resolution, necessitating highly parallel hardware architectures to maintain feasible simulation times. Traditionally, these simulations have been offloaded to Graphics Processing Units (GPUs). However, Field-Programmable Gate Arrays (FPGAs) have emerged as a compelling alternative. @?? FPGAs offer distinct advantages over GPUs, including deterministic latency, superior energy efficiency [3], and the ability to implement highly customizable memory architectures for parallelism. Recent research has shown that through customized memory architectures, FPGAs can outperform GPUs in energy-per-update for structured-mesh solvers [6].

Despite these advantages, FPGAs are traditionally difficult to program using Hardware Description Languages (HDLs). Implementing an FDTD simulation in HDL is tedious and requires the modifications of standard components that are better suited for automation. @?? High-Level Synthesis (HLS) addresses these challenges by allowing designers to describe complex hardware architectures using high-level languages such as C/C++. Recent benchmarking has demonstrated that HLS can achieve performance parity with manual RTL design while significantly accelerating the development cycle [7, 8].

This paper presents an optimized 3D FDTD accelerator designed via HLS, exploring several architectural optimization strategies, including:

Spatial Tiling Maximizing memory bandwidth by utilizing on-chip Block Random Access Memories (BRAMs) to store local stencil and reduce external memory access.

HLS Directives Leveraging compiler pragmas to optimize loop pipelining and unrolling, thereby improving the throughput of the kernel.

Task-Level Parallelism Decomposing the algorithm into concurrent tasks to increase parallelism.

Through these optimization strategies, we demonstrate that an HLS-driven approach can also produce a high-performance FDTD accelerator, while significantly improving code readability, maintainability, and design iteration speed compared to traditional HDL-based workflows.

2 Background

This section establishes the theoretical and technical foundation required to design and implement an FPGA-based 3D FDTD accelerator. To provide a complete context for the methodology, the discussion begins with the governing physics of electromagnetic wave propagation through Maxwell's Equations. Subsequently, it details the transition from continuous calculus to a discrete numerical model using Yee's staggered grid and the resulting leapfrog update equations. Finally, the section explores the architectural characteristics of Field-Programmable Gate Arrays (FPGAs) and the High-Level Synthesis (HLS) design flow. This hardware background is essential for understanding how algorithmic bottlenecks are addressed through spatial tiling and parallel execution, which are the primary focus of this work.

2.1 Maxwell's Equation

The underlying physics governing electromagnetic waves are described by the interaction of electric and magnetic fields, a theory unified by J. C. Maxwell [9]. Maxwell consolidated the independent laws of Faraday, Ampère, and Gauss, demonstrating that electric and magnetic fields are interdependent. While the original theory consisted of 20 equations, it was later refined by O. Heaviside into the modern four-vector representation. These equations, representing Gauss's Law, Gauss's Law for Magnetism, Faraday's Law, and Ampère's Law, are given by:

$$\nabla \cdot \mathbf{D} = \rho_v \tag{1a}$$

$$\nabla \cdot \mathbf{B} = 0 \tag{1b}$$

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \tag{1c}$$

$$\nabla \times \mathbf{H} = \frac{\partial \mathbf{D}}{\partial t} + \mathbf{J} \tag{1d}$$

where the constitutive relations for linear, isotropic media are:

$$\mathbf{D} = \varepsilon \mathbf{E} \tag{1e}$$

$$\mathbf{B} = \mu \mathbf{H} \tag{1f}$$

$$\mathbf{J} = \sigma \mathbf{E} \tag{1g}$$

@??

While (1) provides a complete description of electromagnetism, it is more practical for numerical wave propagation analysis to substitute the constitutive relations directly into the curl equations. By substituting the flux densities $(\mathbf{D}, \mathbf{B})$ and the current density $(\mathbf{J})$ into the curl equations, we obtain the coupled system of first-order partial differential equations:

$$\nabla \times \mathbf{E} = -\mu \frac{\partial \mathbf{H}}{\partial t} - \sigma_m \mathbf{H} \tag{2a}$$

$$\nabla \times \mathbf{H} = \varepsilon \frac{\partial \mathbf{E}}{\partial t} + \sigma \mathbf{E} \tag{2b}$$

This representation shows explicitly the interaction between the electric $(\mathbf{E})$ and magnetic $(\mathbf{H})$ field intensities, providing the fundamental framework for the FDTD method's time-stepping procedure. By rearranging (2) to isolate the temporal derivatives, we obtain the form used for numerical integration:

$$\frac{\partial \mathbf{H}}{\partial t} = -\frac{1}{\mu} \left(\nabla \times \mathbf{E} + \sigma_m \mathbf{H} \right) \tag{3a}$$

$$\frac{\partial \mathbf{E}}{\partial t} = \frac{1}{\varepsilon} \left(\nabla \times \mathbf{H} - \sigma \mathbf{E} \right) \tag{3b}$$

2.2 The FDTD Method

To solve the coupled system of equations in (3) using a digital computer, the continuous temporal and spatial derivatives must be transformed into a discrete form. The Finite-Difference Time-Domain (FDTD) method, first proposed by Kane S. Yee in 1966 [2], achieves this by employing central-difference approximations. This approach maps the electric and magnetic fields onto a discrete staggered grid, allowing the system to be solved iteratively through a "leapfrog" time-stepping procedure. To ensure the numerical stability of this iterative scheme, the time step must satisfy the Courant-Friedrichs-Lewy (CFL) condition [10].

2.2.1 Yee's Staggered Grid

To numerically solve Maxwell's equations, the continuous spatial and temporal domains must be discretized. The FDTD method utilizes the Yee staggered grid, an arrangement where electric (**E**) and magnetic (**H**) field components are spatially interleaved within a unit cell, as shown in Figure 1, where the **E** field and **H** field are staggered by half a temporal step.

In this configuration, each **E** component is located at the center of the edges of a grid cell, while each **H** component is positioned at the center of the faces. The spatial coordinates of the field components within a single Yee cell are summarized in Table 1.

Table 1: Spatial offsets for **E** and **H** field components.

Axis	**Electric Field (E)**	**Magnetic Field (H)**
x	$(i+\frac{1}{2}, j, k)$	$(i, j+\frac{1}{2}, k+\frac{1}{2})$
y	$(i, j+\frac{1}{2}, k)$	$(i+\frac{1}{2}, j, k+\frac{1}{2})$
z	$(i, j, k+\frac{1}{2})$	$(i+\frac{1}{2}, j+\frac{1}{2}, k)$

@??

This spatial staggering ensures that each field component is surrounded by four circulating dual-field components. This layout provides a natural representation of the curl operators, enabling second-order accurate central-difference approximations. From a hardware perspective, this arrangement creates a 3D stencil dependency; updating a single point requires values from its neighbors, which necessitates efficient on-chip memory management to maintain high throughput.

While the Yee grid defines the spatial distribution of the fields, the FDTD method also requires discretization in the temporal domain. To achieve a stable and accurate simulation, a "leapfrog" time-stepping scheme is employed. As illustrated in Figure 1, the magnetic fields are offset by half a temporal step relative to the electric fields, allowing the interleaved calculation of field updates.

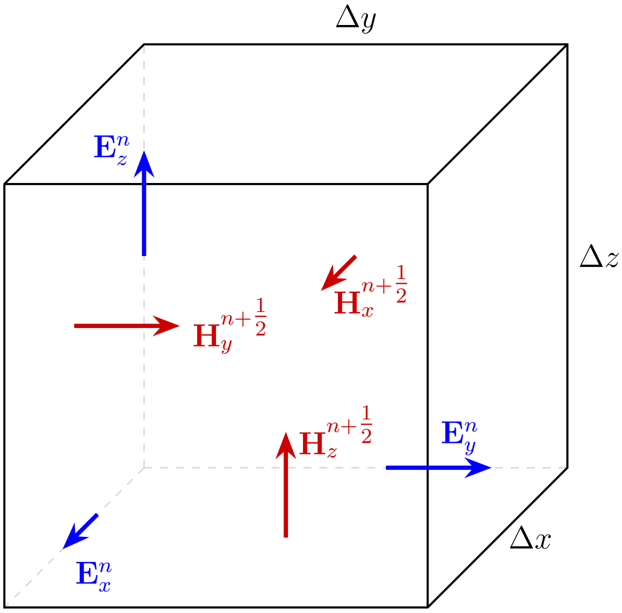
Δy
Δz
Δx
$\mathbf{E}_z^n$
$\mathbf{H}_x^{n+\frac{1}{2}}$
$\mathbf{H}_y^{n+\frac{1}{2}}$
$\mathbf{H}_z^{n+\frac{1}{2}}$
$\mathbf{E}_y^n$
$\mathbf{E}_x^n$

Figure 1: Spatial arrangement of electric ($\mathbf{E}$) and magnetic ($\mathbf{H}$) field components. Dimensions $\Delta x, \Delta y, \Delta z$ represent grid increments, and n denotes the temporal step.

2.2.2 Temporal Update Equations

In addition to spatial staggering, the FDTD method employs temporal interleaving, commonly referred to as the "leapfrog" scheme as described by Taflove [1]. Under this arrangement, the $\mathbf{E}$ components are updated at integer time steps $(n, n+1, \dots)$, whereas the $\mathbf{H}$ components are updated at half-integer time steps $(n + 1/2, n + 3/2, \dots)$.

By applying central-difference approximations to the temporal and spatial derivatives in Maxwell's Equations, we obtain the explicit update equations. For a general lossy medium, the update for a single component such as H_x is expressed in Equation 4:

$$\begin{aligned}\mathbf{H}_x^{n+1/2}[i,j,k] &= \left(\frac{1-\frac{\sigma_m \Delta t}{2\mu}}{1+\frac{\sigma_m \Delta t}{2\mu}}\right)\mathbf{H}_x^{n-1/2}[i,j,k] \\ &+ \left(\frac{\frac{\Delta t}{\mu}}{1+\frac{\sigma_m \Delta t}{2\mu}}\right)\left[\frac{\mathbf{E}_y^n|_{k+1}-\mathbf{E}_y^n|_k}{\Delta z} - \frac{\mathbf{E}_z^n|_{j+1}-\mathbf{E}_z^n|_j}{\Delta y}\right]\end{aligned} \tag{4}$$

In the case of a lossless, homogeneous medium such as free space, the conductivities σ and σ_m are zero. This reduces the decay coefficients to unity, and the update equations for the full 3D system can be simplified to:

$$\mathbf{H}_x^{n+1/2} = \mathbf{H}_x^{n-1/2} + \frac{\Delta t}{\mu_0}\left[\frac{\delta\mathbf{E}_y^n}{\Delta z} - \frac{\delta\mathbf{E}_z^n}{\Delta y}\right] \tag{5a}$$

$$\mathbf{H}_y^{n+1/2} = \mathbf{H}_y^{n-1/2} + \frac{\Delta t}{\mu_0}\left[\frac{\delta\mathbf{E}_z^n}{\Delta x} - \frac{\delta\mathbf{E}_x^n}{\Delta z}\right] \tag{5b}$$

$$\mathbf{H}_z^{n+1/2} = \mathbf{H}_z^{n-1/2} + \frac{\Delta t}{\mu_0}\left[\frac{\delta\mathbf{E}_x^n}{\Delta y} - \frac{\delta\mathbf{E}_y^n}{\Delta x}\right] \tag{5c}$$

$$\mathbf{E}_x^{n+1} = \mathbf{E}_x^{n} + \frac{\Delta t}{\varepsilon_0}\left[\frac{\delta\mathbf{H}_z^{n+1/2}}{\Delta y} - \frac{\delta\mathbf{H}_y^{n+1/2}}{\Delta z}\right] \tag{5d}$$

$$\mathbf{E}_y^{n+1} = \mathbf{E}_y^{n} + \frac{\Delta t}{\varepsilon_0}\left[\frac{\delta\mathbf{H}_x^{n+1/2}}{\Delta z} - \frac{\delta\mathbf{H}_z^{n+1/2}}{\Delta x}\right] \tag{5e}$$

$$\mathbf{E}_z^{n+1} = \mathbf{E}_z^{n} + \frac{\Delta t}{\varepsilon_0}\left[\frac{\delta\mathbf{H}_y^{n+1/2}}{\Delta x} - \frac{\delta\mathbf{H}_x^{n+1/2}}{\Delta y}\right] \tag{5f}$$

where δ denotes the central-difference operator (e.g., $\delta\mathbf{E}_y^n = \mathbf{E}_y^n|_{k+1} - \mathbf{E}_y^n|_k$, where the index $+1$ refers to the spatial increment along the corresponding axis).

This representation reveals that the field update is a function of the previous state and the surrounding spatial curl, scaled by a material-dependent ratio ($\Delta t/\mu_0$ or $\Delta t/\varepsilon_0$). These ratios, combined with the grid spacing Δs, form the basis for the normalized coefficients used in hardware acceleration.

2.3 Field-Programmable Gate Array (FPGA) Architecture

While the 3D FDTD algorithm possesses significant inherent symmetry and parallelism, its performance is highly dependent on the ability of the underlying hardware to exploit these features. Traditionally, Graphics Processing Units (GPUs) have dominated this field due to their massive SIMD (Single Instruction, Multiple Data) parallelism. However, FPGAs have emerged as a powerful alternative for accelerating computational electromagnetics.

The primary advantage of an FPGA-based design over a GPU lies in its architectural flexibility. Unlike the fixed memory hierarchy of a GPU, an FPGA allows the designer to implement a fully customized memory architecture, as noted in [6]. This capability is essential for 3D FDTD, as it enables the creation of application-specific data paths and local memory buffers that can exploit the spatial dependencies of the Yee stencil. By tailoring the hardware to the algorithm, FPGAs can achieve high throughput with lower power consumption and deterministic latency. As noted by Zohouri et al. [4], the advantage of FPGAs in stencil-based high-performance computing lies in their ability to exploit fine-grained parallelism to achieve high performance-per-clock ratios.

To implement the complex update equations defined in Equation (5), the FPGA fabric utilizes several specialized hardware primitives:

- **Configurable Logic Blocks (CLBs):** Consisting of Look-Up Tables (LUTs) and Flip-Flops (FF), which are the fundamental building blocks of the FPGA fabric. They implement the "glue logic" and control flow of the system. In an HLS-driven design, LUTs are used to build the Finite State Machines (FSMs) that coordinate dataflow between BRAM buffers and DSP slices. An increase in available LUTs allows for greater control complexity and more sophisticated optimization strategies.

- **Digital Signal Processing (DSP) Slices:** These are dedicated silicon blocks optimized for high-speed arithmetic. Each slice typically consists of a pre-defined set of operations, allowing field updates to be performed in a single clock cycle without consuming vast amounts of general-purpose logic. An increase in DSP count directly translates to higher throughput, as more field components can be updated in parallel. Architecture studies in [5] demonstrate that the spatial parallelism of FPGAs is uniquely suited to the concurrent field updates of the 3D FDTD grid.

- **Block RAM (BRAM):** These represent the on-chip memory resources that distinguish FPGAs from fixed-cache architectures. Unlike standard global memory (DRAM), BRAM can be "partitioned" and "reshaped" to provide multiple independent access ports. This is a critical feature for stencil computations, where the hardware must fetch several neighboring values simultaneously. Increasing BRAM capacity allows larger grid volumes to be stored on-chip, reducing the need for high-latency external memory access. Recent methodologies have focused on automatically extracting these memory patterns to reduce the overall FPGA memory footprint for complex stencil codes [11].

The performance and scalability of an FPGA-based FDTD accelerator are directly constrained by the availability of these resources. Since the 3D FDTD algorithm is both computationally and memory-intensive, there is a fundamental trade-off between the simulation volume and the update speed. A higher count of DSP slices allows for more Processing Elements (PEs) to operate in parallel, while an abundance of BRAM enables the storage of larger grid dimensions on-chip, thereby avoiding the significant latency penalties associated with external DDR memory access. Consequently, an optimized design must balance the utilization of these resources to maximize the throughput, measured in Millions of cells per second (Mcells/s), while remaining within the physical limits and memory bandwidth constraints of the target device.

2.4 High-Level Synthesis (HLS)

While FPGA architectures provide the necessary primitives for acceleration, manual implementation of large-scale projects in low-level Hardware Description Languages (HDLs) is labor-intensive and error-prone. Many hardware structures, such as memory controllers and nested loop pipelines, follow standard patterns that are better suited for automation than manual coding. High-Level Synthesis (HLS) addresses these challenges by acting as an abstraction layer that automates the generation of these standard structures [12]. Furthermore, HLS allows designers to describe complex projects using high-level languages like C or C++, significantly lowering the barrier of entry compared to RTL design in Verilog or VHDL.

To bridge the gap between algorithmic description and hardware realization, HLS provides several critical optimization directives and data-handling paradigms [12]:

- **Pipelining (`#pragma HLS PIPELINE`):** This is a fundamental optimization for FDTD. It enables the concurrent execution of different stages of the update equation. By pipelining the inner loops, the hardware can begin a new field update before the previous one has completed, with the primary objective of achieving an Initiation Interval (II) of 1.

- **Memory Partitioning (`#pragma HLS ARRAY_PARTITION`):** Standard BRAM is limited by its physical port count (typically two). Since 3D FDTD stencils require multiple simultaneous operands, HLS allows for the "partitioning" of these arrays into smaller, independent memory banks. This provides the parallel access ports necessary to feed the DSP update engines without stalling the pipeline, a technique widely utilized to overcome the memory bottlenecks of structured-mesh solvers [6, 11].

- **Task-Level Parallelism (`#pragma HLS DATAFLOW`):** This directive enables the execution of functions in a producer-consumer pattern. In the context of this work, it allows the electric and magnetic field update functions to operate concurrently, overlapping their execution to maximize the utilization of the FPGA's spatial resources.

- **Streaming Interfaces (`hls::stream` or `hls::stream_of_blocks`):** Beyond standard memory-mapped access, HLS supports a streaming data paradigm through built-in libraries (`hls_streamofblocks.h` or `hls_stream.h`). Using FIFO (First-In-First-Out) buffers, data can be passed point-to-point between hardware modules without the overhead of global memory addressing. In 3D FDTD, streaming is essential for moving planes of data through the update engine, minimizing latency and enabling the high-throughput "Stream of Blocks" architecture [13].

By utilizing these directives, HLS enables a "stepwise refinement" design process. A designer can begin with a baseline software implementation and iteratively apply hardware constraints to transform a sequential C-program into a high-performance, parallelized 3D FDTD accelerator. As shown in automation frameworks such as SASA [14], overlapping computation and communication optimizations are critical for scaling FDTD accelerators to handle massive data volume of 3D electromagnetic simulations.

3 Methodology

The development of the 3D FDTD accelerator followed a process of stepwise refinement, transitioning the initial algorithmic model into a hardware-optimized accelerator. This progression was driven by the specific memory and computational bottlenecks identified during the synthesis of the 3D Yee stencil. The following subsections detail the evolution from a baseline implementation to an advanced streaming architecture, exploring the specific hardware-software co-design strategies, such as domain decomposition, memory partitioning, and task-level parallelism, that were employed to maximize the system's throughput.

@??

3.1 Baseline Implementation

The design process began with a baseline implementation where the 3D FDTD update equations, as defined in Equation (5), were ported to HLS using a standard software-based approach [15]. In this initial stage, the 3D grid components were represented as flattened one-dimensional (1D) arrays in global memory to simplify the interface between the host and the accelerator. Accessing specific spatial coordinates (i, j, k) required manual index linearization, typically in the form `idx = (i * NY + j) * NZ + k`. To establish a performance baseline, the `#pragma HLS PIPELINE II=2` directive was applied to the nested loops of the update functions.

While this approach is standard in software-based FDTD implementations, it introduced significant architectural overheads within the HLS environment. Specifically, the use of flattened 1D arrays necessitated complex index linearization, where every field access required multiple integer multiplications and additions just to calculate a memory address. This resulted in an inefficient allocation of hardware resources, as valuable DSP and LUT slices were consumed by indexing logic rather than the primary field update computations.

Furthermore, the 3D stencil dependency requires field values from neighboring planes at $z+1$ and $z-1$ offsets. In a flattened 1D array, these neighbors are stored at large address intervals, creating a non-contiguous access pattern that inherently restricts the compiler's ability to utilize efficient burst memory transactions. As a result, the HLS scheduler is unable to achieve single-cycle throughput, necessitating a higher Initiation Interval (II) to resolve these distant data dependencies.

To circumvent these memory-mapped bottlenecks, an attempt was made to transition the design toward a streaming data paradigm. However, initial attempts to integrate streaming interfaces (`hls::stream`) directly into this baseline revealed significant architectural conflicts. Specifically, the naive integration of streaming without prior memory management led to severe resource contention. Synthesis reports for a simplified "pass-through" version of the kernel, designed only to read and write data, indicated a latency of upwards of 3 million clock cycles. The full design size report illustrating this surge is provided in Appendix A. This unacceptably high count was a result of the compiler's inability to resolve the 3D stencil dependencies across a sequential stream. These results confirmed that a more sophisticated domain decomposition and memory subsystem strategy was required before high-throughput streaming could be successfully implemented.

3.2 Domain Decomposition

To resolve the architectural limitations of the 1D baseline, a domain decomposition strategy was implemented. This approach was inspired by the parallel hardware architecture proposed by Kong and Su [5], specifically their specialized data mapping used to resolve 3D stencil dependencies. While their work demonstrates an ultra-optimized 1D mapping, this design adopts an intermediate 2D Z-plane decomposition to ensure architectural clarity and more predictable synchronization within the HLS framework.

The Z-dimension ($N_z = 70$) was specifically selected as the primary axis of decomposition because it represents the largest spatial extent of the target grid. To maximize memory efficiency, the implementation was refactored to be Z-major, ensuring that entire 2D horizontal slices are stored contiguously in memory. This alignment allows the accelerator to perform high-speed burst reads from global memory, streaming entire planes into the FPGA's on-chip buffers with minimal addressing overhead.

In this strategy, the 3D computational volume is processed as a series of these horizontal Z-planes, as illustrated in Figure 2. As established by the Yee algorithm in Section 2.2, updating field components at a specific Z-coordinate (plane k) requires the values from the adjacent planes ($k - 1$ and $k + 1$). By explicitly buffering these specific planes in on-chip BRAM, the accelerator can satisfy these 3D dependencies while only interacting with a small, localized subset of the total grid at any given time.

This architectural shift effectively resolved the scheduler complexity that led to the "instruction surge" documented in Appendix A. By narrowing the HLS compiler's scope to a 2D planar update, the tool was able to synthesize an optimized state machine within the device's logic limits. Furthermore, the localized interaction within BRAM provided the necessary foundation for the memory partitioning and task-level throughput optimizations discussed in the subsequent sections.

3.3 Memory Subsystem Optimization

While the Z-plane domain decomposition provided a structural foundation for the accelerator, the initial implementation remained limited by the physical constraints of the FPGA's on-chip memory. Several strategies were explored to alleviate these bottlenecks, specifically focusing on refactoring the data layout and maximizing memory port availability through hardware banking.

3.3.1 Data Layout: Flattened vs. Multidimensional Arrays

The initial baseline utilized flattened 1D arrays where spatial coordinates were accessed via manual index linearization, such as `ex[y * NX + x]`. While common in software-based implementations, this approach obscures the spatial relationship of the data from the HLS compiler. By transitioning to multidimensional C-arrays (`ex[y][x]`), the structural intent of the 3D stencil was made explicit. This clarity allowed the compiler to better infer data dependencies and enabled more sophisticated hardware-level memory manipulations, such as cyclic partitioning, that are not feasible with raw pointers and linearized offsets.

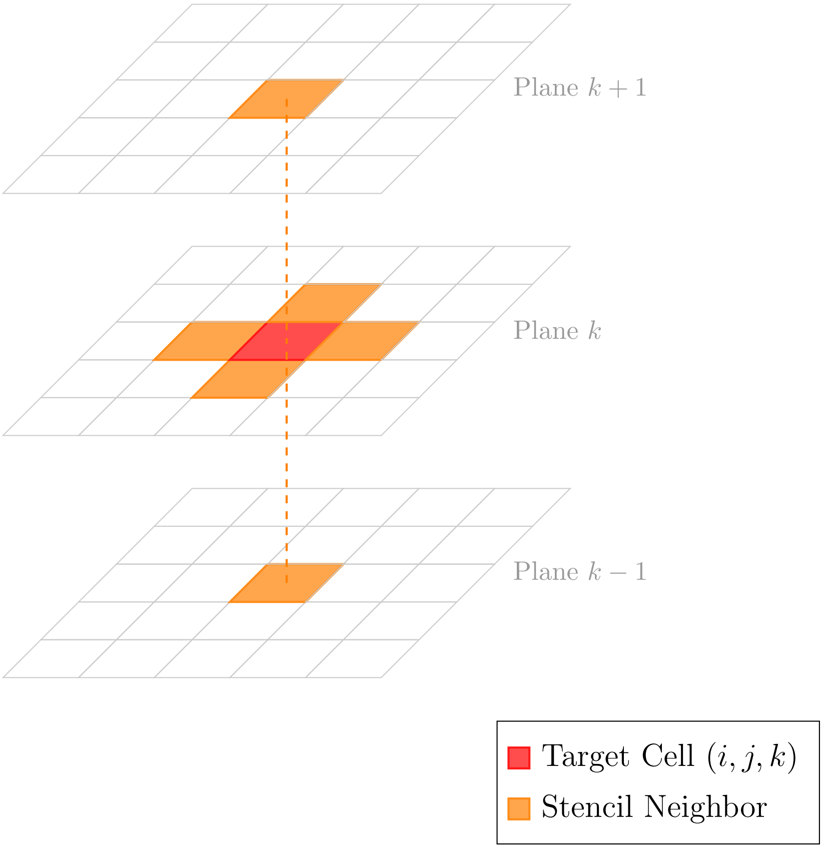

Figure 2: Z-plane memory mapping of the FDTD 7-point stencil. The target field update (Red) requires parallel access to neighboring cells (Orange) across the current plane and the adjacent $k \pm 1$ buffers stored in BRAM.

3.3.2 Memory Port Parallelism: Array Partitioning

As established in Section 2.2, the FDTD stencil requires simultaneous access to multiple field components. Standard BRAM primitives, however, are physically limited to two independent access ports, which forces the HLS scheduler to serialize memory fetches and stalls the pipeline. To resolve this "port starvation", the `#pragma HLS ARRAY_PARTITION` directive was applied to the Z-plane buffers. This optimization splits a single logical array into multiple independent hardware memory banks. This provided the parallel ports necessary to retrieve all stencil operands in a single clock cycle, satisfying the throughput requirements of the update engine without necessitating complex manual memory management.

3.3.3 Task Granularity: Functional Decomposition

To further refine the hardware schedule, the monolithic update loops were decomposed into modular functions, as shown in Listing 1. By applying the `#pragma HLS INLINE off` directive, the synthesis tool was directed to treat each field update as a dedicated hardware module. This "divide and conquer" approach allowed the HLS compiler to optimize the internal pipeline of each field independently. Furthermore, this modularity served as a critical prerequisite for overlapping execution, enabling the scheduler to manage the distinct computational requirements of the electric and magnetic field updates as separate processes rather than a single, oversized task.

```
static void update_HX(float hx_plane[NY_1][NX_0],
                      float ey_plus1[NY_1][NX_0],
                      float ey_plane[NY_1][NX_0],
                      float ez_plane[NY_0][NX_0]) {
#pragma HLS INLINE off

  // HX
  for (int y = 0; y < NY_1; ++y) {
    for (int x = 0; x < NX_0; ++x) {
#pragma HLS PIPELINE II = 1
      const float dey = ey_plus1[y][x] - ey_plane[y][x];
      const float dez = ez_plane[y + 1][x] - ez_plane[y][x];
      hx_plane[y][x] += ch * (dey - dez);
    }
  }
}

// similar code for update_HY and update_HZ

```

```
static void update_H_crit(float hx[NY_1][NX_0],
                          float hy[NY_0][NX_1],
                          float hz[NY_1][NX_1],
                          float ex[NY_0][NX_1],
                          float ex1[NY_0][NX_1],
                          float ey[NY_1][NX_0],
                          float ey1[NY_1][NX_0],
                          float ez[NY_0][NX_0]) {
#pragma HLS INLINE off
  update_HX(hx, ey1, ey, ez);
  update_HY(hy, ez, ex1, ex);
  update_HZ(hz, ex, ey);
}
```

Listing 1: Implementation of the H-field update engine demonstrating Functional Decomposition

3.3.4 Buffer Management: Global vs. Local Scope

The final refinement in this stage involved refactoring the scope and lifecycle of the buffers. Initially, Z-plane buffers were declared as `static` global variables, which created rigid hardware bindings that restricted the scheduling flexibility of the compiler. By transitioning to local, function-scoped buffers passed explicitly as arguments between the `read`, `compute`, and `write` stages, the data dependencies were made transparent to the HLS tool. This shift allowed the compiler to more effectively manage the transition between external memory-mapped access and high-speed internal BRAM updates.

3.4 Advanced Throughput Optimization

While the memory optimizations described in the previous section resolved port contention and local dependencies, the accelerator still operated as a sequential state machine. In this model, the hardware must fully complete the reading of a plane, the computation of the update, and the write-back to external memory in a strict linear order. While this process is efficient when pipelined internally, significant throughput gains can be achieved by overlapping these high-level tasks. By utilizing task-level parallelism, the design can begin reading the next plane while the current one is still being computed. The following subsections detail the transition from a sequential dataflow to an advanced streaming architecture, exploring the trade-offs between point-to-point streaming and block-based synchronization.

3.4.1 Temporal Overlapping: Ping-Pong Buffers

The first attempt at task-level parallelism utilized the concept of **Ping-Pong Buffer** (double buffering) to overlap the memory access and computation stages. By declaring the field arrays with an additional bank dimension (`hx_plane[2][NY][NX]`), the HLS scheduler was instructed via the `#pragma HLS DATAFLOW` directive to process these banks concurrently. In this model, while the memory stage populates one bank, the compute engine could simultaneously process the other. This strategy aimed to obscure the high latency of global memory transfers by ensuring that the update engines were never starved of data.

3.4.2 Robust synchronization: `hls::stream_of_blocks`

While manual ping-pong arrays provide a theoretical throughput increase, they often introduce complex synchronization challenges and potential pointer-aliasing errors with the HLS scheduler. To resolve these issues, the design transitioned to the `hls::stream_of_blocks` paradigm.

Unlike standard sequential streams, `hls::stream_of_blocks` allows for the transmission of an entire 2D plane as an atomic hardware block. This paradigm provided two specific advantages for the 3D FDTD accelerator. First, it enabled implicit handshaking, where the HLS framework automatically manages the ownership of the BRAM buffers between the producer and consumer functions. This automation eliminated the manual indexing and pointer-aliasing errors encountered in the initial ping-pong implementation.

Furthermore, it ensured random access compatibility, which is crucial for the FDTD 7-point stencil. Because `stream_of_blocks` permits the compute engine to perform random-access reads within the transmitted block, the hardware could efficiently fetch the $i \pm 1$ and $j \pm 1$ neighbors required for the update logic. This capability provided the necessary data-access flexibility for the stencil math without the massive logic overhead that would be required to reconstruct spatial neighborhoods from a traditional sequential FIFO stream.

4 Evaluation

4.1 Environments

4.2 Results

5 Discussion

6 Conclusion

References

[1] A. Taflove and S. C. Hagness, *Computational Electromagnetics: The Finite-Difference Time-Domain Method*, 3rd ed. Boston, MA: Artech House, 2005, the definitive reference for FDTD theory and implementation.

[2] K. S. Yee, “Numerical solution of initial boundary value problems involving maxwell’s equations in isotropic media,” *IEEE Transactions on Antennas and Propagation*, vol. 14, no. 3, pp. 302–307, 1966.

[3] T. Nguyen *et al.*, “FPGA-based HPC accelerators: An evaluation on performance and energy efficiency,” *Concurrency and Computation: Practice and Experience*, vol. 34, no. 18, 2022.

[4] H. R. Zohouri *et al.*, “High-performance computing with FPGAs: Case study in optimizing stencil kernels,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 5, pp. 975–989, 2018.

[5] C. Kong and T. Su, “Parallel hardware architecture of the 3d FDTD algorithm with convolutional perfectly matched layer boundary condition,” *IEEE Transactions on Antennas and Propagation*, vol. 64, no. 9, pp. 3936–3946, 2016, provides a comprehensive analysis of the parallel data paths required for 3D FDTD.

[6] K. Kamalakkannan, “High-level FPGA accelerator design for structured-mesh-based numerical solvers,” Ph.D. dissertation, University of Warwick, 2023. [Online]. Available: https://wrap.warwick.ac.uk/180089/

[7] Y. Bai *et al.*, “Towards a comprehensive benchmark for high-level synthesis targeted to FPGAs,” in *37th Conference on Neural Information Processing Systems (NeurIPS)*, 2023.

[8] G. Rodriguez-Canal, N. Brown *et al.*, “Stencil-hmls: A multi-layered approach to the automatic optimisation of stencil codes on FPGA,” in *Proceedings of the SC ’23 Workshops*, 2023.

[9] J. C. Maxwell, *A Treatise on Electricity and Magnetism*, 1st ed. Oxford: Clarendon Press, 1873.

[10] R. Courant, K. Friedrichs, and H. Lewy, “On the partial differential equations of mathematical physics,” *IBM Journal of Research and Development*, vol. 11, no. 2, pp. 215–234, 1967.

[11] R. Szafarczyk, S. W. Nabi, and W. Vanderbauwhede, “Reducing FPGA memory footprint of stencil codes through automatic extraction of memory patterns,” in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, 2022.

[12] Advanced Micro Devices, Inc., *Vitis HLS User Guide*, v2025.2 ed., AMD, November 2025, uG1399.

[13] H. K.-H. So *et al.*, “Throughput-oriented FPGA acceleration of FDTD,” in *2018 International Conference on Field-Programmable Technology (FPT)*, 2018.

[14] X. Tian, Z. Ye, A. Lu, L. Guo, Y. Chi, and Z. Fang, “SASA: A scalable and automatic stencil acceleration framework for optimized hybrid spatial and temporal parallelism on HBM-based FPGAs,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 16, no. 2, pp. 1–33, 2023, provides a framework for optimizing 3D stencils by balancing spatial and temporal parallelism.

[15] J. B. Schneider, *Understanding the Finite-Difference Time-Domain Method.* World Color, 2010, a comprehensive resource for the practical implementation of FDTD. [Online]. Available: https://eecs.wsu.edu/~schneidj/ufdtd/

Appendices

A Extended Insight into Baseline Implementation with Streaming

The following report illustrates the internal compiler results for the initial attempt at synthesizing a streaming-based 3D FDTD engine without memory optimization or domain decomposition. This implementation was designed as a "pass-through" test, focusing solely on the logic required to read from and write to global memory.

As highlighted in Table 2 the final `HW Transforms` phase, step (2) `optimizations`, the final instruction count reached a critical surge of 3,309,857. This count can be attributed to the HLS scheduler being forced to map the entire three-dimensional mesh simultaneously to satisfy the dataflow constraints, leading to a massive expansion of the state-machine logic. This diagnostic output served as the primary justification for the architectural shift toward the Z-plane domain decomposition described in Section 3.2.

Table 2: Vitis HLS Design Size Report for the Naive Dataflow implementation, exhibiting the instruction surge discussed in Section 3.1.

```
================================================================
== Design Size Report
================================================================

* Total Instructions per Compilation Phase
+---------------+----------------------------+--------------+----------------------------------------------------------------------------------------+
| Phase         | Step                       | Instructions | Description                                                                            |
+---------------+----------------------------+--------------+----------------------------------------------------------------------------------------+
| Compile/Link  |                            |      360     | After all functions are compiled and linked into a single design                       |
|               |                            |              |                                                                                        |
| Unroll/Inline |                            |              | After user unroll and inline pragmas are applied                                       |
|               | (1) unroll                 | 2,482,290 *  | user unroll pragmas are applied                                                        |
|               | (2) simplification         | 2,482,254 *  | simplification of applied user unroll pragmas                                          |
|               | (3) inline                 | 2,482,260 *  | user inline pragmas are applied                                                        |
|               | (4) simplification         | 2,482,260 *  | simplification of applied user inline pragmas                                          |
|               |                            |              |                                                                                        |
| Array/Struct  |                            |              | After user array partition and struct aggregate/disaggregate pragmas are applied       |
|               | (1) array partition        | 2,482,278 *  | user array partition pragmas are applied                                               |
|               | (2) simplification         | 2,482,278 *  | simplification of applied user array partition & struct aggregate/disaggregate pragmas |
|               | (3) aggregate/disaggregate | 2,482,278 *  | user struct aggregate/disaggregate pragmas are applied                                 |
|               | (4) array reshape          | 2,482,278 *  | apply array reshape pragmas                                                            |
|               | (5) access patterns        | 1,654,982 *  | array access pattern optmizations                                                      |
|               |                            |              |                                                                                        |
| Performance   |                            |              | After transformations are applied to meet performance pragma targets                   |
|               | (1) loop simplification    | 1,654,982 *  | loop and instruction simplification                                                    |
|               | (2) parallelization        | 1,654,982 *  | loops are unrolled or pipelined to meet performance targets                            |
|               | (3) array partition        | 1,654,982 *  | arrays are partitioned to meet performance targets                                     |
|               | (4) simplification         | 1,654,982 *  | simplification of design after performance transformations                             |
|               |                            |              |                                                                                        |
| HW Transforms |                            |              | After hardware transfomations                                                          |
|               | (1) lowering               | 1,655,060 *  | initial conversion to HW specific instructions                                         |
|               | (2) optimizations          | 3,309,857 *  | high level synthesis optimizations                                                     |
+---------------+----------------------------+--------------+----------------------------------------------------------------------------------------+
    * - Exceeded design size warning message threshold

* Instructions per Function for each Compilation Phase
+---------------------+---------------+--------------+---------------+--------------+-------------+---------------+
| Function            | Location      | Compile/Link | Unroll/Inline | Array/Struct | Performance | HW Transforms |
+---------------------+---------------+--------------+---------------+--------------+-------------+---------------+
| + fdtd              | strem.cpp:366 | 360          | 2,482,260 *   | 1,654,982 *  | 1,654,982 * | 3,309,857 *   |
|  + read_to_stream   | strem.cpp:332 | 109          | 1,241,113     | 827,471      | 827,471     | 1,654,895     |
|     gmem0_to_stream | strem.cpp:212 |  17          | 205,344       | 136,906      | 136,906     | 273,805       |
|     gmem1_to_stream | strem.cpp:222 |  17          | 205,344       | 136,906      | 136,906     | 273,805       |
|     gmem2_to_stream | strem.cpp:232 |  17          | 201,810       | 134,550      | 134,550     | 269,093       |
|     gmem3_to_stream | strem.cpp:242 |  17          | 208,320       | 138,890      | 138,890     | 277,773       |
|     gmem4_to_stream | strem.cpp:252 |  17          | 208,320       | 138,890      | 138,890     | 277,773       |
|     gmem5_to_stream | strem.cpp:262 |  17          | 211,968       | 141,322      | 141,322     | 282,637       |
|  + write_to_gmemd   | strem.cpp:349 | 163          | 1,241,113     | 827,471      | 827,471     | 1,654,913     |
|     stream_to_gmem0 | strem.cpp:272 |  26          | 205,344       | 136,906      | 136,906     | 273,808       |
|     stream_to_gmem1 | strem.cpp:282 |  26          | 205,344       | 136,906      | 136,906     | 273,808       |
|     stream_to_gmem2 | strem.cpp:292 |  26          | 201,810       | 134,550      | 134,550     | 269,096       |
|     stream_to_gmem3 | strem.cpp:302 |  26          | 208,320       | 138,890      | 138,890     | 277,776       |
|     stream_to_gmem4 | strem.cpp:312 |  26          | 208,320       | 138,890      | 138,890     | 277,776       |
|     stream_to_gmem5 | strem.cpp:322 |  26          | 211,968       | 141,322      | 141,322     | 282,640       |
+---------------------+---------------+--------------+---------------+--------------+-------------+---------------+
    * - Exceeded design size warning message threshold

* Design Size Message Settings
+---------------------------------------------+--------+-------------------------------------------------------------------+
| Message Setting                             | Value  | Description                                                       |
+---------------------------------------------+--------+-------------------------------------------------------------------+
| config_compile -design_size_maximum_warning | 100000 | Show a warning when total design instructions exceeds this value |
+---------------------------------------------+--------+-------------------------------------------------------------------+
```