fdtd-z: A systolic scheme for GPU-accelerated nanophotonic simulation

Jesse Lu* Jelena Vučković

April 28, 2023

Abstract

fdtd-z is a high-throughput simulation engine designed to enable photonic workflows such as device characterization, manual parameter search, and free-form inverse design — which are typically bottlenecked by the lack of available simulation throughput in solving Maxwell's equations in three dimensions. *fdtd-z* thus aims to provide a photonic simulation engine that

- features state-of-the-art simulation throughput,
- is easily scalable to hundreds of nodes and beyond,
- is open-source, license-free, and widely accessible.

fdtd-z seeks to accomplish this by providing an efficient mapping of the finite-difference time-domain (FDTD) algorithm to graphics processing units (GPUs), integrating with highly parallelizable model training frameworks, and being distributed as open-source software.

At its core, the main innovation that *fdtd-z* provides is a systolic schedule for computing the FDTD update equations specifically designed to overcome the bandwidth limitations of the GPU memory hierarchy. In this whitepaper, we briefly motivate why a systolic scheme is needed, devise a scheme by examining the dependency structure of the FDTD update, and present the numerical results of our implementation.

1 The limited memory hierarchy of the GPU motivates a systolic update scheme

In the last decade, the graphics processing unit (GPU) has gone far beyond a computational engine for visual processing to become the computational tool *de rigeur* in a variety of fields including artificial intelligence, distributed blockchain, and scientific computing. In seeking to do the same for solving Maxwell's equations in the time domain, we first present an

*jesselu@spinsphotonics.com

ultra-simplified view of a modern (Nvidia) GPU as shown in figure 1, which illustrates its massively parallel design.

But although the GPU offers significant computational throughput, figure 1 also reveals the correspondingly limited amount of data throughput offered by its memory hierarchy as detailed in table 1. The picture is particularly glum when considering the general form of the finite-difference time-domain (FDTD) update equations, which features a ratio of ~ 1 in terms of bytes transferred to floating point operations (FLOPS) for the generous case of an already sophisticated implementation — far more than what any level of the GPU memory hierarchy can provide apart from register shuffle transfers.

For this reason, we pursue a systolic scheme (figure 2), where only boundary values are communicated in a hierarchical fashion, in the hope that the decreased bandwidth requirements at lower levels will map more efficiently to the memory hierarchy of the GPU. Critically, *fdtd-z* attempts to do this without modifying the underlying update equations and instead only seeking to find a schedule for the updates which minimizes the number of data transfers performed.

2 Devising an efficient, systolic schedule for the FDTD update

The fundamental dependency structure of the FDTD update is most easily visualized for the case of only one spatial dimension (figure 3). From this simple picture, we can already see the difficulty in decreasing the amount of data transfers needed per update (figure 4) as well as the inefficiencies incurred by the natural solution of using halos (figure 5).

In contrast, we suggest three alternatives to halos which avoid redundant loads and updates:

- *span* (not pictured): a brute-force strategy which eliminates spatial dependencies by simply loading all nodes to local memory,
- *skew* (figure 6): rotating the computational cell which avoids circular dependencies,
- *scan* (figures 7 and 8): reusing previously computed values, allowing for a flattened computational cell.

These three techniques are combined to realize an update schedule for the full three-dimensional FDTD update that avoids circular dependencies without incurring the overhead of using halos.

First, the *span* technique is used along the z-axis. Practically, although this limits the z-extent of the simulation domain to ~ 100 unit cells, the majority of nanophotonic structures are of subwavelength dimension in the out-of-plane axis and can fit comfortably within this constraint. Next, the *skew* technique is used to eliminate circular dependencies along the y-axis, resulting in the diamond-shaped computational cell shown in figure 9. Finally, the

scan technique is used along the x-axis (figures 10 and 11) which results in the need to only load and store values along the leading and trailing edges of the cell.

The consequence of such an update scheme is that inter-diamond communication can be formalized as reading from and writing to unidirectional buffers (figure 12). As such, communication only occurs "downwind" — there are no circular dependencies.

Another consequence of adopting a diamond-shaped computational cell to a rectilinear simulation domain is pictured in figure 13, which also illustrates the natural solution of breaking up the simulation domain into communication buffers, wrap-around scanning with an increment along the y-axis, and a temporal skew of the simulation domain along the x-y direction. Because the scanning and buffers both wrap around the x- and y-boundaries of the simulation domain the entire domain may now be repeatedly updated in a truly systolic manner (figure 14) in that the computational cells never need to have their contents reset or reloaded.

This update scheme is not only systolic but is also hierarchical, because any diamond cell can itself be repeatedly decomposed into smaller diamonds interconnected with analogous communication buffers (figure 15), allowing for a one-to-one mapping of the FDTD update algorithm to the hierarchical-systolic commucation strategy proposed in figure 2.

3 Preliminary numerical results for a practical FDTD code implementing the systolic scheme

We implement the hierarchical-systolic scheme outlined here in a practical FDTD code which features the following design choices:

- *Limited z-extent of simulation domain.* As previously described, dependencies along the z-axis are dealt with using the *span* technique. In our implementation, this limits the extent to ~ 100 unit cells along the z-axis.
- *Perfectly-matched layers (PML) along z-axis only.* As a direct result of spanning the z-axis, PML layers, which require a more complex update equation as well as extra auxiliary fields, are only implemented along z.
- *Adiabatic absorption profiles allowed in the x-y plane.* Instead of PML layers, aborption profiles which do not vary in z, are used.
- *Source profile in the x-y plane.* Only a source consisting of in-phase and quadrature components for a single x-y plane is considered for the purposes of numerical results.

The basic arithmetic and data requirements on a per-thread level for this implementation (table 2) would have us expect to still be bandwidth-limited by a factor of ~ 2.5, which is what we find when executed on a Nvidia Quadro RTX4000 GPU which is theoretically capable of 14.2 TFLOPS of 16-bit floating-point performance. A computate-limited implementation would then achieve a performance of ~ 0.21 TCUPS (68 floating-point operations

per cell update); in practice, we achieve a real-world performance of ~ 0.05 TCUPS which is a signal that the code should still be able to be optimized further. Such optimization can probably be most easily realized by using the latest Nvidia GPUs (i.e. H100) and leveraging the newly-introduced features in CUDA 12.

Lastly, we note that in order to simplify our analysis, we have neglected to include the overheads associated with the temporal skew of the simulation domain, the extra cells needed for the PML and absorber boundary conditions, as well as other smaller factors which also vary based on the details of the simulation setup (number of timesteps, size of simulation domain, PML thickness, ...).

4 Conclusion

A high-throughput, open-source, and scalable photonic simulation engine has immense potential to extend, and even transform, the realm of what is possible to achieve with nanophotonic devices today. To this end, we have presented the design and preliminary numerical results for *fdtd-z*, a mapping of the FDTD update algorithm to the massively parallel architecture of the GPU in the form of a systolic update scheme.

This work has been supported by the ARPA-E DIFFERENTIATE program.

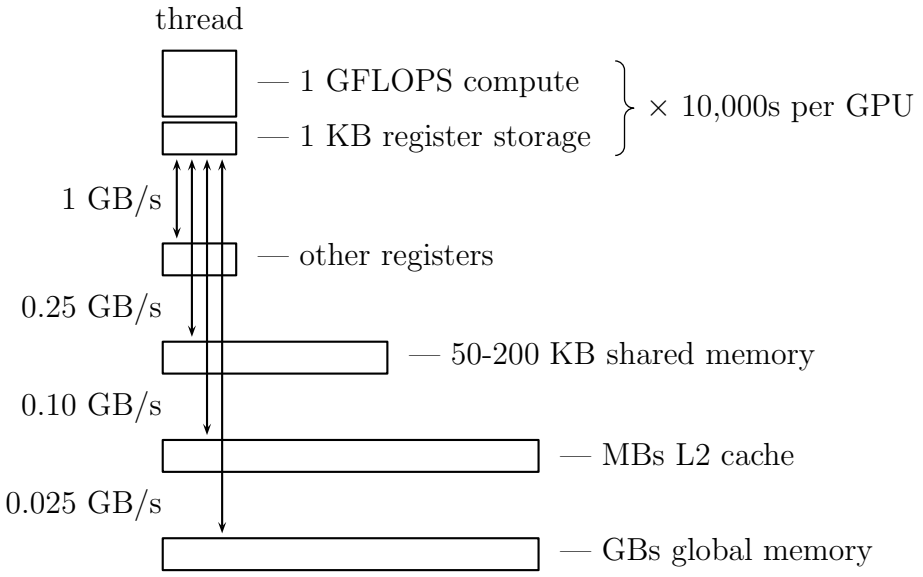

Figure 1: Drastically simplified model of a GPU. Modern GPUs have tens of TFLOPS (10^{12} floating-point operations) of processing power distributed across 10,000s of parallel execution units, or threads. While each thread can perform $\sim 10^9$ floating-point operations per second, all input and output values are required to reside in a register bank consisting of only ~ 1 KB of storage. Although a hierarchy of data channels of varying bandwidths, sizes, and accessibility, as outlined in this figure and detailed in table 1, exists to transfer data to and from registers, efficiently utilizing this memory hierarchy for a computationally-sparse operation such as the FDTD update algorithm (which naively requires at least 1 byte of data transfer per floating-point operation) remains non-trivial.

Data channel	GB/s/thread	Size	Accessibility	Overhead
Register (shuffle)	1.00	up to 32 KB	32 threads	none
Shared memory	0.25	50-200 KB	256 threads	small
L2 cache	0.10	1-100 MB	all threads	large
Global memory	0.025	10-100 GB	all threads	large

Table 1: Simplified view of GPU data hierarchy. *Register memory* is directly accessible and physically adjacent to the individual compute cores of a GPU. While the register data channels are designed to match the throughput of the compute cores, it is also possible to perform register-to-register shuffle operations within groups of 32 threads (a thread warp). Thread warps are inherently synchronized and shuffle operations carry no synchronization overhead. *Shared memory* is physically located at the streaming multiprocessor (SM) level and is accessible to all threads within a thread block (8 warps, or 256 threads, assuming 1 KB of register storage per thread on a CUDA-capable GPU). Shared memory transfers are slower than register shuffles and incur a small synchronization overhead within the thread block (between thread warps). The *L2 cache* is a single pool of on-chip storage which is accessible by any thread. Use of the cache is generally not explicitly programmable and often carries with it a large overhead associated with synchronizing all of the threads on the GPU. *Global memory* is the primary memory resource, in that it is the only one which persists across GPU kernel execution. It resides off-chip, features the smallest bandwidth channel, and is often the primary bottleneck in kernel execution.

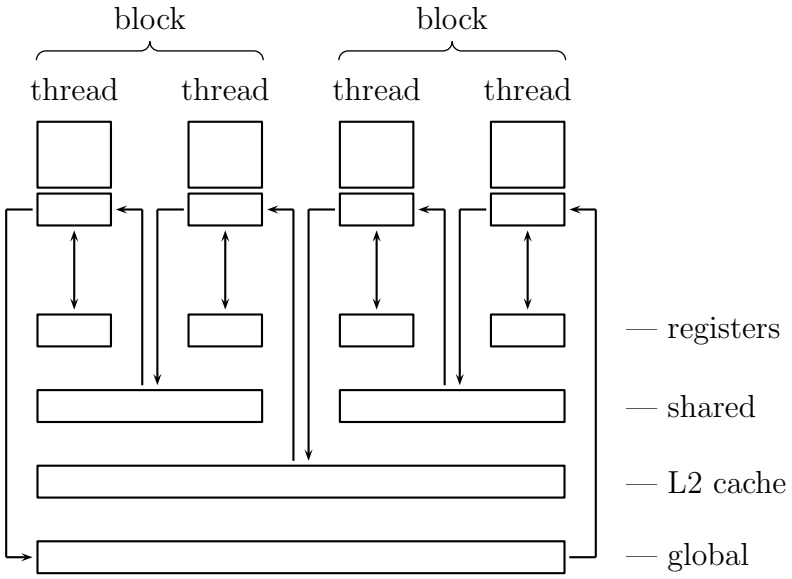

Figure 2: Systolic use of the GPU memory hierarchy. The bandwidth limitations of the GPU require that an efficient mapping of the FDTD update algorithm to not only be able to perform multiple updates for each field value loaded, but that the ratio of updates-per-load be increasing as one moves down the memory hierarchy. One way this can be accomplished is with a systolic scheme where each level of the memory hierarchy is responsible only for communication across a corresponding thread boundary. This works because the lower levels of the memory hierarchy, which feature smaller bandwidths and larger overheads, are now only responsible for communication across coarser thread boundaries and thus experience a larger number of updates per load at their level, as desired. *fdtd-z* features a systolic scheme which uses 1) warp-shuffle operations to implement data transfers along the z-axis of the simulation domain, 2) shared memory access to allow for lateral communication across warp boundaries within the same thread block, 3) L2 memory to transfer data across thread block boundaries, and 4) global memory access only for loading and storing field values at the leading and trailing edges, respectively, of the entire thread grid. A critical feature of the systolic scheme used by *fdtd-z* is that all communication (except for shuffle operations along the vertical axis) is one-directional, which, among other advantages, allows for the large synchronization overhead associated with transferring data across L2 and global memory to be partially hidden with a buffering scheme.

	arithmetic	register	shared	L2	global
FDTD update	960	128	–	–	–
field buffer	–	–	240	120	24
material buffer	–	–	120	30	6
absorption profile	–	96	–	–	4
source profile	64	128	–	–	4
PML update	64	256	–	–	–
total	1088 operations	608 bytes	360 bytes	150 bytes	38 bytes

Table 2: Average per-step arithmetic operations and data transfers per thread in the implemented systolic scheme. When compared to table 1, the bandwidth requirements for *fdtd-z* roughly match the capabilities of the GPU. In this simple analysis, arithmetic operations are counted in terms of fused multiply-add (FMA) instructions, where each FMA instruction is counted as 2 floating-point operations, even when either the multiply or addition is not needed. Each thread updates 16 cells per-step (2 layers of 8 cells each), where each of the 6 components per cell requires 10 floating-point operations, for a total of 960 operations. The basic FDTD update also requires register shuffle transfers to communicate values between threads along the z-axis. The data transfers needed for the field and material buffers are computed for the case of the Nvidia Quadro RTX4000 GPU which allows for a 6×6 ($u \times v$) array of diamonds in the hierarchical-systolic scheme, each consisting of a 2×4 array of per-thread diamonds. In addition to the unidirectional buffers needed to communicate field and material values, absorption and source profiles which only vary in the x-y plane are also needed. Lastly, register shuffles are also needed to communicate the auxiliary values between cells found within the perfectly-matched layer (PML) boundary conditions, and are used for the absorption and source values as well.

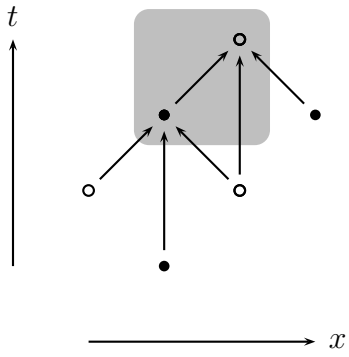

Figure 3: Dependency structure for a single Yee cell in the 1D FDTD algorithm. Updating the Yee cell in the dotted square requires 4 field values (two E-field and two H-field values) and corresponds to a CUPL (cell-updates-per-load) of 0.25.

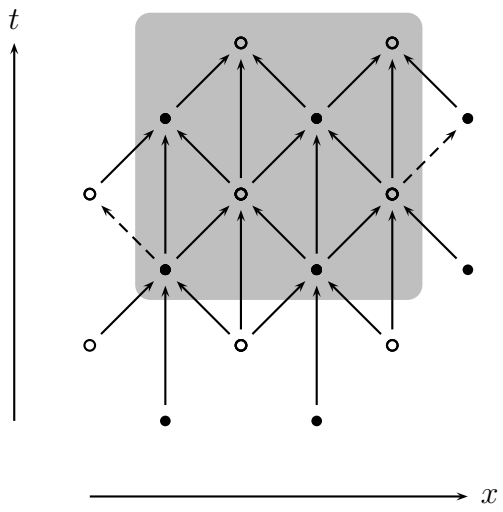

Figure 4: Naive dependency structure of a 2×2 block of Yee cells. A 2×2 block of Yee cells is dependent on 8 field values. Such a scheme corresponds to an improved memory-efficiency of 0.5 CUPL when compared with the baseline of figure 3. Generally, a $M \times N$ block has a CUPL of $\frac{MN}{2(M+N)}$, or $N/4$ for the case where $M = N$. In practice, the $4N$ loads (when $M = N$) will need to be stored in a finite amount of scratchpad memory, $S = 4N$, where S is the number of field values which can be stored in scratchpad memory. Under this constraint, the resulting storage-limited performance is $S/16$ CUPL. Unfortunately, a naive tiling of the simulation domain in such blocks is not possible because of circular dependencies between adjacent blocks. This occurs at the left and right block boundaries which not only contain incoming dependencies but also feature outgoing dependencies (dashed arrows).

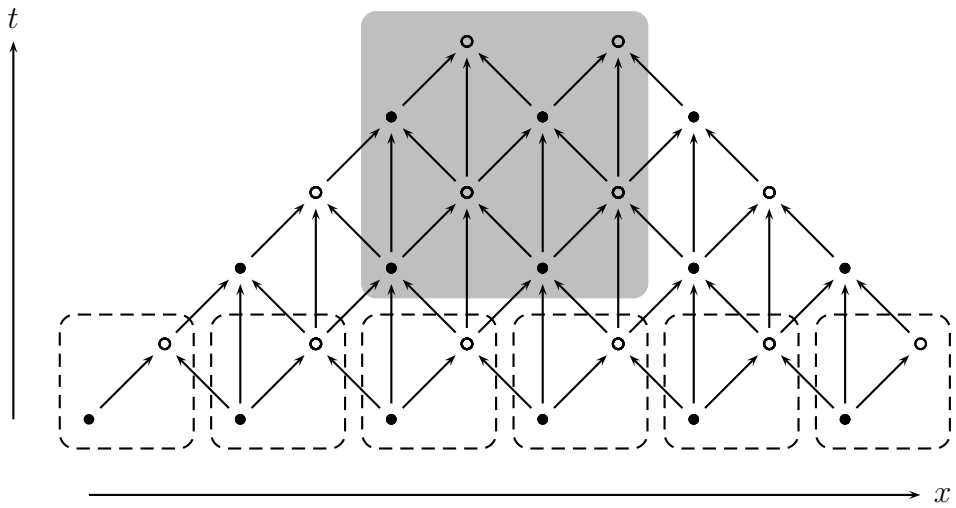

Figure 5: Dependency structure of a 2×2 block with halo. A 2×2 block updated exclusively from the field values of previous timesteps requires a total of 6 Yee cells (dashed boxes) to be loaded. The haloing strategy incurs the penalty of duplicated loads and computation in the overlapping regions of adjacent tiles in exchange for avoiding the problematic circular dependencies encountered in figure 4. In general, updating a $N \times N$ block with halos requires $6N - 1$ field values to be loaded, resulting in a performance of $\sim N/6$ CUPL. In practice, the size of the block is capped by the fact that required number of loads must fit in scratchpad memory ($S = 6N - 1$) which results in a storage-limited performance of $\sim S/36$ CUPL.

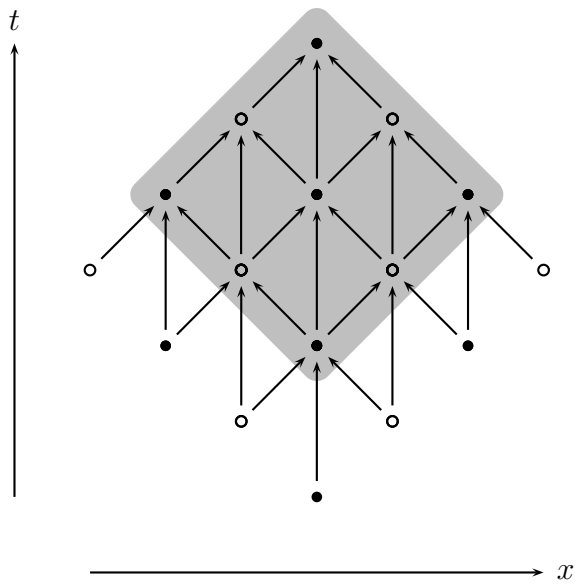

Figure 6: Dependency structure of a "diamond" block. A diamond-shapred block containing N^2 field values requires $2N + 1$ loads and exhibits $\sim N/4$ CUPL (1 cell update is equivalent to 2 field value updates). The storage-limited performance of $\sim S/8$ CUPL is not only superior to that of the halo strategy in figure 5, but even to the case where circular dependencies exist, as in figure 4. This presents a 45-degree rotation, or skewing, of the computational cell as an enticing alternative to the halo technique, avoiding circular dependencies without the need for redundant loads or computation. Intuitively, the advantages of skewing can be thought of as simply arising from a natural alignment of the computational cell with the underlying dependency structure of the finite-difference grid, which exhibits a diamond-like pattern.

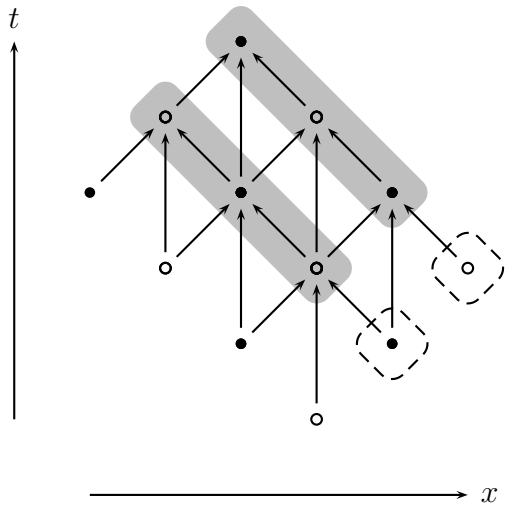

Figure 7: Dependendency structure of a diamond block scanned along x-t. In addition to halo and skew techniques, a scanning technique can also be used to increase performance by not only flattening the computational cell along one dimension and allowing it to expand along the others, but also by reusing previously computed nodes to eliminate the number of loads needed for the current iteration. Here, we show that the scanning technique applied to the diamond block from figure 6 results in the ultimate updates-per-load performance of needing only a single load for a theoretically unlimited number of updates. Practically, of course, the number of updates will be limited by the size of the scratchpad, giving a storage-limited performance of $S/2$ CUPL.

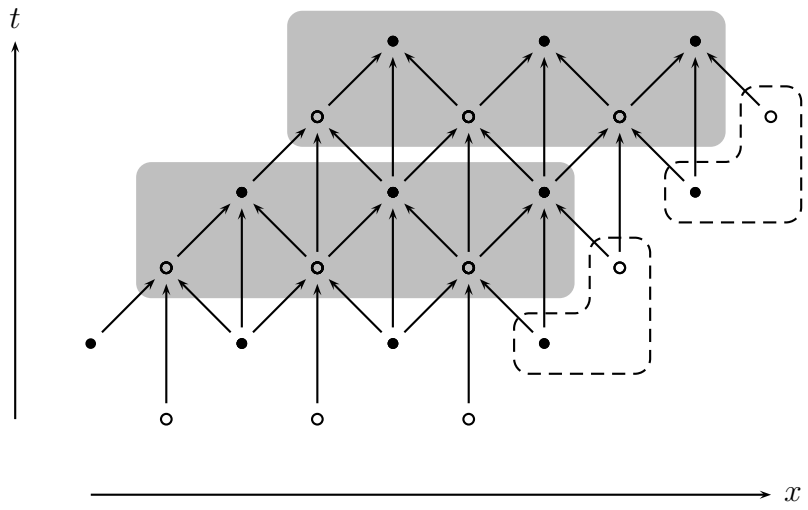

Figure 8: Dependency structure of a $1 \times N$ block scanned along x-t. Scan technique applied to a simplified computational cell consisting of a linear array of Yee cells. Although the computational cell is flattened along the t dimension, scanning still occurs along the x-t diagonal which serves to avoid circular dependencies and the need for a halo. Compared to figure 7, performance decreases to $S/4$ CUPL because two loads are now needed to update the computational cell; however, the fact that all E- and H-field nodes are located at the same time step greatly simplifies the practical implementation of this scheme, especially when working with the full three-dimensional update. For this reason, the implementation of *fdtd-z* likewise uses a computational cell that is flattened along t, and that is scanned along the x-t diagonal.

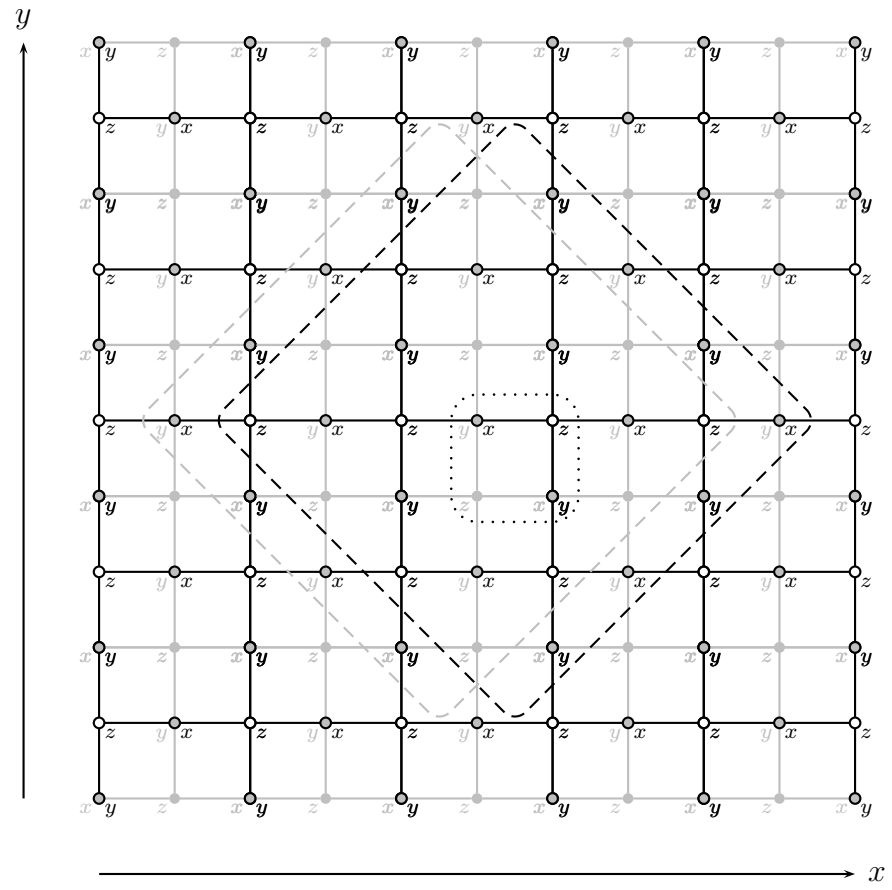

Figure 9: Diamond computational cell used along x-y plane. For the full three-dimensional problem scan, skew, and span techniques are used for the x-, y-, and z-axes respectively. This allows for the z-axis to be "flattened", and for the computational cell to have a diamond-like representation in the x-y plane. Critically, such a computational cell completely tiles the simulation domain while avoiding any circular dependencies. We specifically picture a computational cell of size $N = 4$ here, where N corresponds to the number of Yee cells along the main diagonals of the diamond. In general, a diamond of size N allows for $N^2/2$ cell updates and requires $\sim 6N$ loads which results in a scratchpad-limited performance of $\sim \sqrt{S/72}$ CUPL, comparing favorably against an unskewed scheme using a square cell of side-length N and scanning along the x-y-t diagonal which requires $\sim 12N$ loads to update N^2 cells resulting in only $\sim \sqrt{S/144}$ CUPL. In this figure, empty circles represent H-fields, filled gray circles represent E-fields, and gray circles with a dark outline are points where H- and E-fields are colocated. Corresponding subscripts denote the components of the respective fields. The H- and E-field nodes which are included in the computational cell are those contained in the darker and lighter dashed diamond respectively. For reference, the Yee cell considered to be located at $(x, y) = (0, 0)$ is identified with a dotted square.

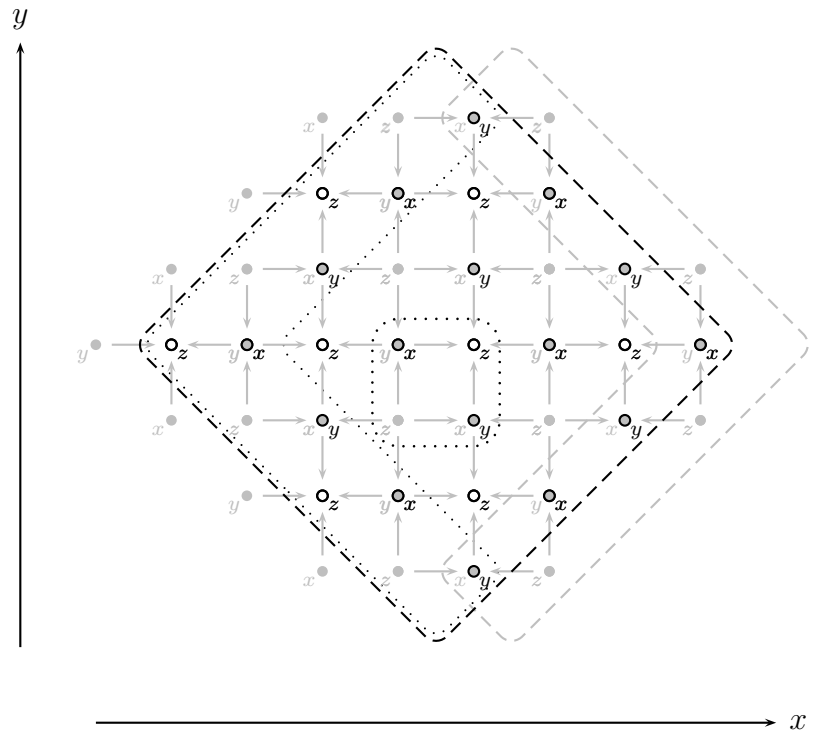

Figure 10: Detail of the H-layer updates, loads, and writes for the diamond cell. The dashed diamond identifies the $3N^2/2$ H-field nodes to update for a diamond of size $N = 4$. We show the $3N - 1$ E-field nodes which need to be loaded within the gray dashed line at the leading edge of the diamond, while the other dependent E-field nodes (which are already included in the accompanying E-field diamond) are shown as well. Lastly, the $3N - 2$ H-field nodes which need to be written out to memory are denoted within dotted lines at the trailing edge of the diamond. The reference Yee cell at $(0, 0)$ is pictured as a black dotted square.

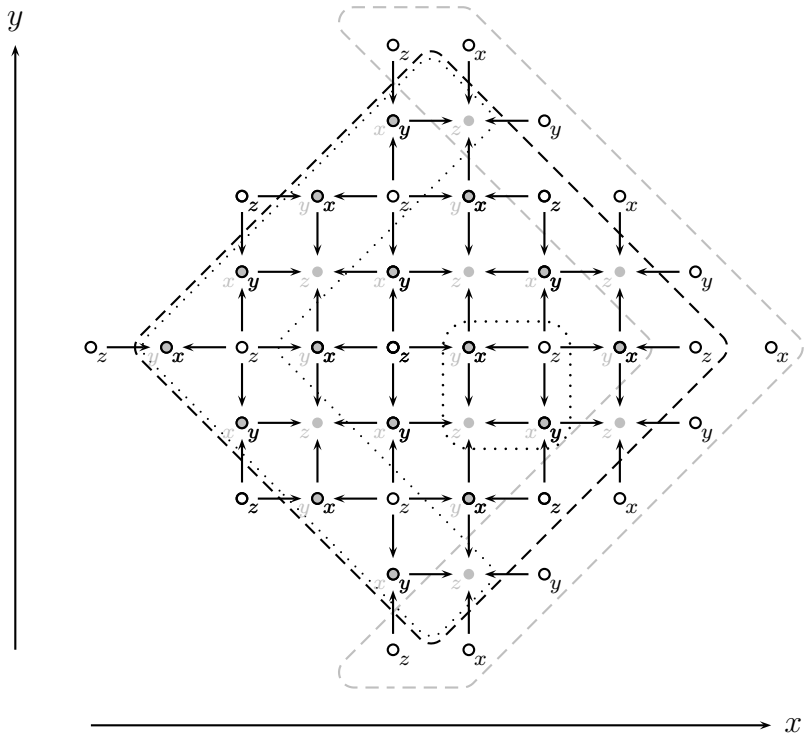

Figure 11: Detail of the E-layer updates, loads, and writes for the diamond cell. Similar to figure 10, the $3N^2/2$ E-field nodes to update, $3N + 2$ H-field nodes to load, and $3N - 1$ E-field nodes to write out are denoted within black dashed, gray dashed, and dotted lines respectively. The reference Yee cell at $(0, 0)$ is once again denoted with a black dotted square.

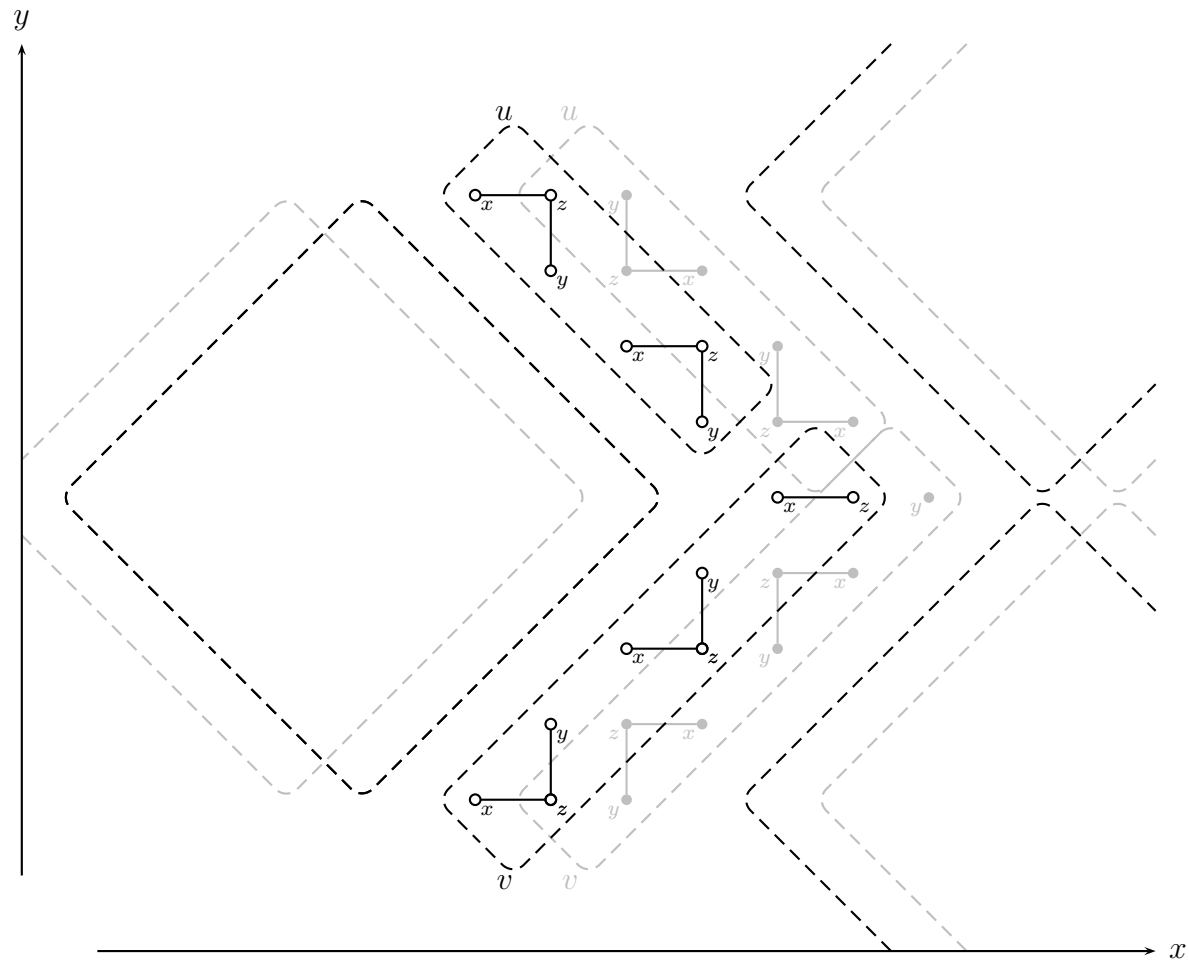

Figure 12: u- and v-buffers used to communicate between diamonds. As the diamond computational cell is scanned in the $+x$ direction, the "downwind" transfer of values occurs by means of buffers which are orthogonal to either the u- or v- axes, defined in (x, y) coordinates as $u = (1, 1)$ and $v = (1, -1)$. Together, both u- and v-buffers contain all the dependent nodes needed to update the downwind computational cell. Here, the buffers for H-field nodes are denoted with black dashed lines while the E-field buffers are denoted with gray dashed lines. The corresponding computational cells for E- and H-fields are denoted corrrespondingly as well. Lastly, note that the v-buffer is selected to hold the H_x, H_z, and E_y nodes at the center of the diamond.

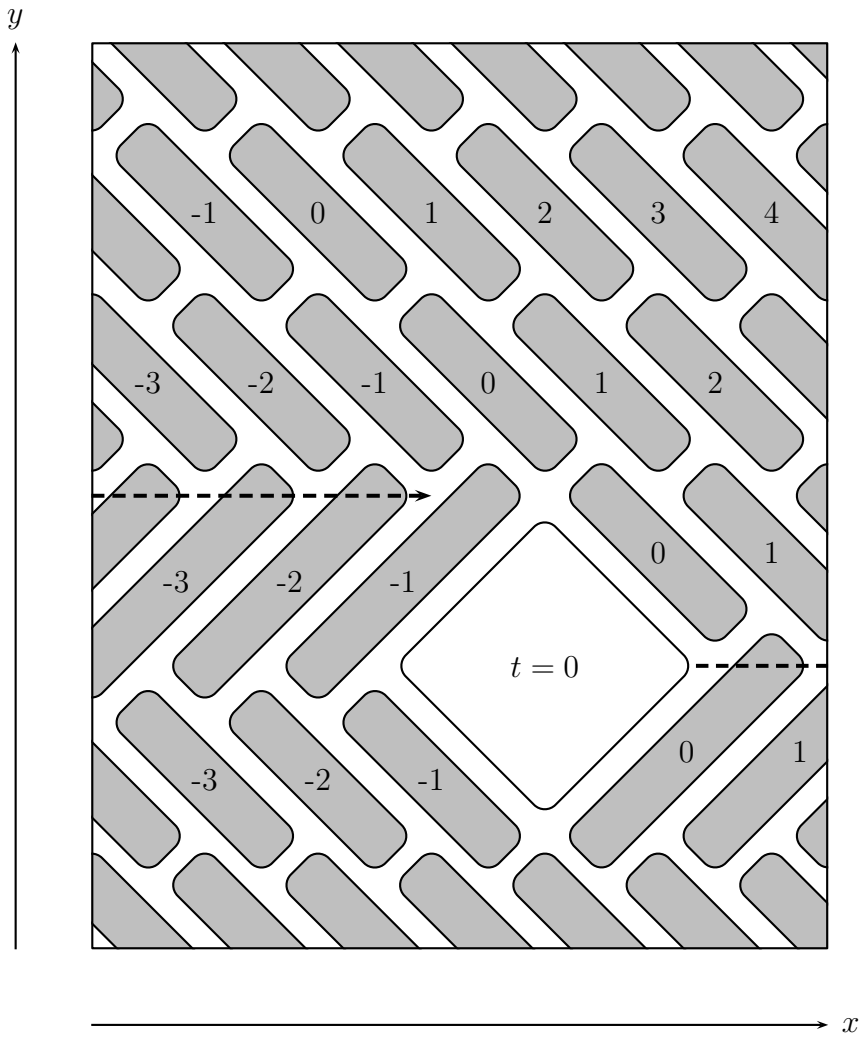

Figure 13: Systolic decomposition of the simulation domain. The decomposition of the simulation domain into computational cells, u-buffers, and v-buffers is shown. The black arrow shows the traversal of diamond computational cell along the x-axis with wrap-around at which point the y-position increases by a half-diamond width. In so doing, the entire simulation is traversed repeatedly (wrap-around boundaries are also used in the y-direction) and in a self-consistent manner, in that u- and v-buffers of matching time step with the diamond are always present at the leading edge of the computational cell, and buffers of the previous time step are emitted at the trailing edges of the diamond. Achieving this in a causal manner, requires that the simulation domain be skewed along the $u = (1, 1)$ direction as can be seen in the upper-right portion of the domain, where the numbers within the gray buffers denote the time step relative to the computational cell.

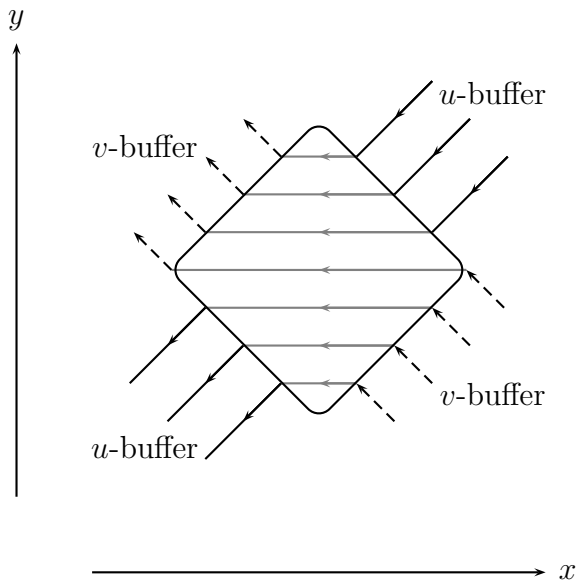

Figure 14: Cell-centric view of the systolic update. The systolic decomposition from figure 13 produces a computational cell where input nodes flow first from the input u-buffer at the top-right, traverse the cell and exit via the v-buffer, return (upon hitting the x-boundary and wrapping around the domain) via the input v-buffer at the lower-right, and exit via the output u-buffer at the bottom-left. Such a view is labeled as "cell-centric" in that the details of scanning across the simulation domain can now be abstracted away, in favor of a model where field values incrementally enter and exit the computational cell. The arrows in the figure show a conceptual representation of how field values enter the computational cell (represented as a diamond) via the u- or v-leading edge, are updated as the cell is scanned forward, and then exit via the u- or v-trailing edge.

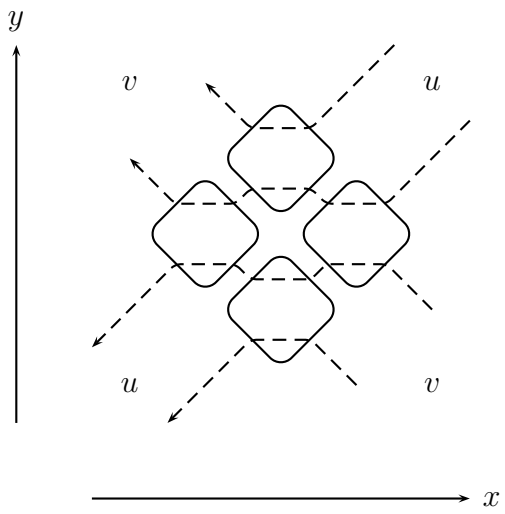

Figure 15: Hierarchical composibility of the systolic computational cell. Illustrates how the systolic cell is composable in a hierarchical way (smaller diamonds can be combined into larger ones, etc...) where systolic cells at each level communicate via u- and v-buffers. Such a design is a fitting match for the systolic design in figure 2 because increasingly higher-levels of the cell hierarchy require fewer values to be transferred per iteration, matching the lower-bandwidth bandwidth levels of the memory hierarchy. Additionally, the unidirectional, downwind nature of the cell-to-cell dependency structure, which is the result of scanning in the x-direction, allows for variable-sized inter-cell buffers which can offset the large synchronization overheads inherent in communicating values via the L2 cache and global memory.