We present a high performance tridiagonal solver library for Xilinx FPGAs optimized for multiple multi-dimensional systems common in real-world applications. A predictive analytical model is developed and used to explore the design space and obtain rapid performance estimates that are 85% accurate. This library achieves an order of magnitude better performance when solving large batches of systems than previous FPGA work. A detailed comparison with a current state-of-the-art GPU library for multi-dimensional tridiagonal systems on an Nvidia V100 GPU shows the FPGA achieving competitive or better runtime and significant energy savings of up to 30%. Through this design, we learn lessons about the types of applications where FPGAs can challenge the current dominance of traditional GPU hardware.

**KEYWORDS**

Tridiagonal Solvers, FPGA, high level synthesis, HPC

**ABSTRACT**

We present a high performance tridiagonal solver library for Xilinx FPGAs optimized for multiple multi-dimensional systems common in real-world applications. A predictive analytical model is developed and used to explore the design space and obtain rapid performance estimates that are 85% accurate. This library achieves an order of magnitude better performance when solving large batches of systems than previous FPGA work. A detailed comparison with a current state-of-the-art GPU library for multi-dimensional tridiagonal systems on an Nvidia V100 GPU shows the FPGA achieving competitive or better runtime and significant energy savings of up to 30%. Through this design, we learn lessons about the types of applications where FPGAs can challenge the current dominance of traditional GPU hardware.

However, such problems also expose the limits of CPU and GPU acceleration due to a number of factors. For such iterative applications, a sequence of lightweight kernels must be controlled by the iteration loop on the CPU, hence requiring kernel input and output data to be moved through GPU global memory. Unrolling multiple iterations of smaller kernels is also problematic as it requires multiple global memory synchronizations. Data reuse in GPUs is primarily via the cache, the performance of which is variable and dependent on the application. FPGAs enable the design of an architecture optimized around the characteristics of the problem, which can result in improved performance. In this paper we evaluate the design of tridiagonal system solver algorithms on modern FPGAs. Our underlying goal is to understand the criteria for a given system solver to be amenable to FPGA acceleration and uncover the limitations and profitability of such accelerators.

Previous work has utilized both low-level hardware description languages [17, 25, 28] and high-level synthesis tools [4, 14–16, 26], developing single system solvers in isolation without a design strategy that can be applied for multiple systems and multiple dimensions in general. It has also not exploited higher-gain optimizations more important for real-world applications. Comparison of performance to traditional architectures such as GPUs, for multi-dimensional tridiagonal systems has also been limited in current literature, minimizing insights into the utility of FPGAs for these applications. In this paper we attempt to bridge this gap with a unifying workflow for FPGA implementation of implicit solvers for real-world multi-dimensional applications. Specifically we make the following contributions:

- We examine the algorithmic trade-offs in FPGA acceleration of multi-solve, multi-dimensional solvers, proposing a design and optimization strategy that optimizes based on problem size, dimensionality, number of systems solved, and data-flow paths required.

- Using this approach, we design a new tridiagonal solver library that can be used in the solution of multi-dimensional applications. The architecture exploits High Bandwidth Memory (HBM) to combine multiple dimension solves and explicit loops, and batched execution of multiple independent solves. We present the optimized design of two non-trivial applications, a 2D and 3D ADI heat diffusion solve, implemented with both single precision (FP32) and double precision (FP64) floating point representations, and a 2D Stochastic-Local Volatility (SLV) model application from the financial computing domain.

- A predictive analytic model is developed to obtain estimates for application runtime, giving insights into the profitability
of implementing the tridiagonal system solvers on Xilinx FPGAs using our design strategy. The model predicts the runtime performance considering system/batch sizes and optimizations applied together with memory requirements and operating frequency. Runtime predictions are within 15% of that achieved on evaluated applications.

- Finally, we show competitive performance on the Xilinx Alveo U280 compared to an HPC-grade Nvidia V100 GPU, for both FP32 and FP64 precision, and much improved efficiency.

To our knowledge the extended workflow, new library, predictive model, and the superior performance demonstrated for the above applications present key innovations, advancing the state-of-the-art. This also showcases an interesting, albeit narrow, class of applications where the FPGA challenges the performance of GPUs, currently the best hardware for direct solution of multi-dimensional tridiagonal systems, both in terms of runtime and energy consumption. We believe this will be particularly valuable in areas such as financial computing, reducing the complexity of the development cycle for these platforms.

The rest of the paper is organized as follows: Section 2 presents a brief overview of tridiagonal solver algorithms. Section 3 presents our proposed design strategy, starting from the basic algorithms, to target FPGA code for the Xilinx Alveo FPGAs. A performance analysis and benchmarking of the FPGA implementations compared to the GPU performance is presented in Section 4. Finally, we briefly look at related work in Section 5 and conclusions are in Section 7.

2 BACKGROUND

Tridiagonal systems arise from the need to solve a system of linear equations as given in equation (1), where \( a_0 = c_{N-1} = 0 \). Its matrix form \( Ax = d \) can be stated as in equation (2).

\[
a_{i}u_{i-1} + b_{i}u_{i} + c_{i}u_{i+1} = d_{i}, \quad i = 0, 1, \ldots, N - 1
\]

\[
\begin{bmatrix}
  b_0 & c_0 & 0 & \cdots & 0 \\
  a_1 & b_1 & c_1 & \ddots & \vdots \\
  0 & a_2 & b_2 & \ddots & \vdots \\
  \vdots & \vdots & \ddots & \ddots & \vdots \\
  0 & 0 & \cdots & a_{N-1} & b_{N-1} \\
\end{bmatrix}
\begin{bmatrix}
  u_0 \\
  u_1 \\
  u_2 \\
  \vdots \\
  u_{N-1} \\
\end{bmatrix}
= \begin{bmatrix}
  d_0 \\
  d_1 \\
  d_2 \\
  \vdots \\
  d_{N-1} \\
\end{bmatrix}
\]

The solution to such systems of equations are well known. The Thomas algorithm [22] (see Algo. 1) carries out a specialized form of Gaussian elimination (assuming non-zero \( b_i \)) providing the least computationally expensive solution, but suffers from a loop carried dependency. It has a time complexity of \( O(N) \).

### Algorithm 1: thomas(a, b, c, d)

1: \( d_0^* \leftarrow d_0/b_0 \)
2: \( c_0^* \leftarrow c_0/b_0 \)
3: for \( i = 1, 2, \ldots, N - 1 \) do
4: \( r \leftarrow 1/(b_i - a_i c_{i-1}^*) \)
5: \( d_i^* \leftarrow r(d_i - a_i d_{i-1}^*) \)
6: \( c_i^* \leftarrow r c_i \)
7: end for
8: for \( i = N - 2, \ldots, 1, 0 \) do
9: \( d_i \leftarrow d_i^* - c_i^* d_{i+1} \)
10: end for
11: return \( d \)

The Spike algorithm [19] decomposes the \( A \) matrix, into \( p \) partitions of size \( m \) to obtain the factorization of \( A = DS \) where \( D \) is a main diagonal block matrix consisting of tridiagonal matrices \( A_1, \ldots, A_p \) and \( S \) is the so called spike matrix. The solution to the system then becomes, \( DSe = d \) where the system \( Dy = d \) can be used to obtain \( y \), and \( Sx = Y \) to obtain \( x \). Since matrix \( D \) is a simple collection of \( A_1 \), each \( A_iY = d_i \) can be solved independently. Solving \( Sx = y \) requires only solving a reduced penta-diagonal system (see Wang et al. [24] for a detailed description). The algorithm therefore operates in three steps: factorization, reduced system solve, and back substitution, where the factorization (LU and UL) has a complexity of \( O(N) \). The reduced system can be solved directly or indeed can be further reduced to a block diagonal system using the truncated-spike variation that ignores the outer diagonals when \( A \) is diagonally dominant. The Spike algorithm is particularly well suited for solving very large systems on traditional architectures.

### Multiple Systems in 2D/3D

Each of the above algorithms specifies the solution of a single tridiagonal system, which is characteristically a one dimensional problem. However, applications of interest are usually 2 or 3 dimensional, where tridiagonal systems are formed by solving along one of the coordinate axes. This leads to a number of independent systems based on the number of discretization points along the other axis. For example a 3D system with \( N_x \times N_y \times N_z \) number of mesh points will have \( N_y \times N_z \) number of tridiagonal systems in the first dimension (each system with size \( N_x \)), \( N_x \times N_z \) in the second (each with size \( N_y \)) and so on. The ADI method, (used in one of the applications in this work), repeatedly solve tridiagonal systems along these different axes. Here, the \( a_i, b_i, c_i \) and \( d_i \) coefficients are calculated for each grid point, in a way that matches the underlying data structure of
the application; data is stored contiguously in either a row-major (Z is contiguous, Y and X are strided) or more commonly a column-major (X is contiguous, Y and Z are strided) format. This poses a challenge for algorithms that then solve multiple tridiagonal systems simultaneously; coefficients for an individual system will be laid out differently, depending on the direction of the solve. This is especially true on traditional architectures such as CPUs or GPUs [13]. An FPGA design must also carefully consider memory performance when solving such multi-dimensional applications.

3 FPGA DESIGN

FPGAs can offer significant performance through the implementation of deeply pipelined custom datapaths. There is no fixed general purpose architecture exploited by software as in a traditional CPU or GPU. Instead, a tailored datapath for the computation is synthesized using a variety of basic circuit elements. These are digital signal processing (DSP) blocks to implement arithmetic operators, look-up-tables (LUTs) and registers, fast on-chip block memories (BRAM/URAM) of kilobit capacities, clock modules, and a rich routing fabric to connect these elements into a large logical architecture. Large FPGAs comprise multiple interposed die with such resources called Super Logic Regions (SLR). The Xilinx U280 has 3 SLRs. Bandwidth within an SLR is extremely high (TB/s) due to the wealth of connections and memory elements, while between SLRs it is limited by the number of silicon interposer connections available. An FPGA board includes much larger external DRAM (32 GB on the U280) and even High Bandwidth Memory (HBM). Managing the movement of data between these different types of memory is key to achieving high computational performance. The introduction of High-Level Synthesis (HLS) tools has reduced the complexity of FPGA programming, where a high-level programming language such as C++/OpenCL can be used with special directives to target the FPGA. However, achieving high performance is still significantly challenging as code needs to be structured to suit the dataflow/pipelined programming style. The key optimizations required to obtain enhanced performance are transformations enabling pipelining, unrolling loops by replicating computational units (CUs), and tiling to improve locality such that data can be reused by fitting to fast on-chip memory. For a good overview of these techniques we refer the reader to De Matteis et al. [9] and the Xilinx HLS programming guide [3].

3.1 Small and Medium Systems Solves

Considering the resources available on an FPGA, a single tridiagonal system solve, using the Thomas algorithm in Algo. 1, would require 4 multiplications, 1 division and 2 subtractions for the forward path and one multiplication and subtraction for the backward path. However, due to dependencies for computing $d_l^t$ and $c_r^t$, each iteration of the forward path will have to be executed serially, incurring the full arithmetic pipeline latency, $l_f (=30$ clock cycles on a Xilinx U280 FPGA), to pass through the forward loop data path. Additionally the backward loop can only start when all iterations of the forward path have been completed, due to the reverse data access where the loop starts from iteration $N-2$. Thus the total latency for solving a single system with the Thomas algorithm would be approximately $l_f \times N + l_b \times N$ clock cycles (assuming $l_b$ cycles is the arithmetic pipeline latency for completing a single iteration of the backward loop). On the other hand, a PCR based single solver implementation would require 4 multiplications, 9 multiplications and 1 division within the inner loop of Algorithm 2. If $l$ is the arithmetic pipeline latency of the inner loop, then the total number of clock cycles for the PCR algorithm, is $(N + l) \times \log N$

Here we assume that the outer loop is executed serially and a fully pipelined inner loop or an inner loop with an initiation interval of one. Given the inner loop iterations are independent, they can be unrolled by some factor $f_{ij} = 2, 3, \ldots$ which will then require $f_{ij} \times$ the resources to implement the inner loop. The total clock cycles will then be $(N / f_{ij} + l) \times \log N$. The outer loop iterations have a dependency and thus cannot be unrolled.

For the Thomas solver, there are $l_f$ clock cycles between consecutive iterations of a single system solve in the forward path. This can be considered as a dependency distance. As such, we could attempt to solve $l_f$ number of tridiagonal systems to fully utilize the forward path circuit pipeline. This can be done by interleaving the iterations of the forward pass loop of the Thomas solver such that iteration 1 of system 1 is input followed by iteration 1 of system 2 and so on, per clock cycle, up to iteration 1 of system $f_{ij}$. In fact selecting a group, $g = \max(l_f, l_b)$, enables $g$ system solves to be interleaved, saturating the pipeline. If there are $B$ number of total tridiagonal systems to be solved, i.e. a batch size of $B$, then the total latency with Thomas can be shown by (3):

$$
(1 + \lfloor B/g \rfloor) \times gN
$$

(3)

Thus for large $B$ the total latency tends to $BN$. This is a characteristic of all $O(N)$ algorithms, which ideally can be pipelined to accept inputs each clock cycle at the cost of differing resource consumption.

For the PCR algorithm, there are no dependencies between iterations of a single system and solving a batch of $B$ systems (by batching the inner loop) incurs the latency in (4):

$$
(BN / f_{ij} + l) \times \log N
$$

(4)

For large $B$, dividing (4) by (3) gives a factor of $\log N / f_{ij}$ pointing to the fact that the batched Thomas solver is $\log N$ times faster than batched PCR, for $f_{ij} = 1$. Thus, to match the Thomas solver latency, a batched PCR implementation needs an unroll factor $f_{ij} = \log N$. However, given that the PCR inner loop has a considerably higher resource requirement compared to the Thomas solver, the batched Thomas solver will always provide better performance for the same area. An exception to this is when the system size, $N$ is large and FPGA on-chip memory becomes the limiting factor. Designs for such cases are discussed in Sec 3.2.

Considering a batched solver based on the spike algorithm, assume each system in the batch is of size $N$. The algorithm creates $m$ blocks and each has LU and UL factorization done in parallel, followed by the pentadiagonal solve and then back-substitution in parallel. This incurs a total latency given by (5):

$$
(1 + \lfloor Bm/g + 1 \rfloor) \times gN / m + mC
$$

(5)

The latency for the factorization for each block (first term), is similar to a Thomas forward and backward solve carried out in an interleaved manner. Although the number of cycles spent on the pentadiagonal reduced system solve is $BmC$ (assuming a linear latency model) and back substitution is $BN$, only the latency for first stage of pentadiagonal solver is added to equation 5 as all three
modules are pipelined. Back substitution does not add additional delay between its inputs and outputs. When $B$ is sufficiently large and stages are pipelined, a latency of $BN$ is achieved. Again this is due to the spike algorithm having a $O(N)$ complexity. However, if $Bmc \geq BN$ then dataflow must stall for some time decreasing throughput. Resource consumption of the LU/UL factorizations requires $3 \times$ the resources for an equivalent Thomas solver and the pentadiagonal solver needs additional resources (again more than an equivalent Thomas solver).

Given the lower resource requirements and profitability of the Thomas algorithm, compared to the other algorithms, we first focus on its optimized batched implementation on an FPGA for system sizes that can fit into on-chip memory. As we are interleaving groups of $g$, the $a_{i-1}, b_{i-1}$ and $c_{i-1}$ values needs to be stored in on-chip memory such that they can be used in subsequent (ith) iterations. For a FP32 implementation we have found that a grouping of 32 is sufficient to effectively pipeline the computation (this is 64 for FP64) on the Xilinx Alveo U280. The forward and backward loops operate in opposite directions and thus a First-In-First-Out (FIFO) buffer cannot be used, rather on-chip addressable memory is used for data movement. The forward and backward loops can be made to operate in parallel when batching a number of system solves, using ping-pong buffers (also called double buffers). With this technique, dual port memory is partitioned into two parts, one being written while the other is read. Once writes (by the forward pass) and reads (backward pass) are completed, read and write halves are swapped. Note that the very first read has to wait until the very first write has completed. Additionally, the technique also doubles the memory requirement compared to using the same memory portion for both read and write. Including the latency for starting the first write to the ping-pong buffer, and writing back the final result to external memory increases the total latency in (3) by $2gN$ to give: $(3 + [B/g]) \times gN$ clock cycles. The total on-chip memory required for a single Thomas solver interleaving $g$ systems can be computed based on the need to store the $a, c, d, c^*, d^*$ and $u$ vectors, where each consumes $2gN$ words in the ping-pong buffers. The total $12gN$ requirement with dual port memory can be satisfied with 6x dual port block RAMs (URAM/BRAM) each with a capacity of $2gN$. Additionally there is a need to store $g$ values of the $i$ – 1th iteration separately, requiring 4 memories with a capacity of $g$ words.

Data transfer from external memory to on-chip memory plays a crucial part in achieving high performance, especially for multi-dimensional solvers such as the 3D ADI heat diffusion application detailed later in this paper. If we consider a 3D application with systems sizes ($N_g$) of 256 in all three dimensions, then a solve along the x-dimension will have YZ $(256 \times 256 \times 256)$ number of systems to be solved, each corresponding to an x-line of system of size 256. Given the data is stored in consecutive memory locations along the x-lines, good memory throughput can be achieved. However to exploit the full memory bandwidth, a larger number of memory ports needs to be used. For the 512-bit memory ports, on the Alveo U280, it is sufficient to saturate the data-flow pipeline with a width of 256-bits at a 300MHz clock speed, which is our target frequency. This enables us to fetch data sufficient to feed 8 Thomas solvers in parallel. Such a configuration can be viewed as a vectorized Thomas solver. Additionally, the total YZ number of x-lines can be set up to be solved in groups (g) of 32. Here, the 1st Thomas solver circuitry solves the 0th, 8th, 16th and so on x-lines, the 2nd solves 1st, 9th, 17th and so on x-lines, and so on. Batches of x-lines can be solved in such interleaved groups to saturate the data flow pipeline to achieve higher throughput.

In the x-dimension, the reads from external memory bring in data stored in consecutive memory locations. However, the data fetched belongs to the same line (i.e. same system), thus we need to buffer 8 x-lines internally and carry out an $8 \times 8$ transpose to feed that to 8 different solvers (see Figure 1(a) for an illustration of the issue with a $4 \times 4$ transpose). For solving along the y-dimension, we fetch each XY plane to on-chip memory to avoid strided memory accesses and then read along the y-lines from the on-chip memory (see Figure 1(b)). Similarly for solving along the z-dimension, we read in x-lines (which are consecutive in memory) along the z dimension, fetching XZ planes, to on-chip memory. No transpose is required for y- and z-dimension solves as each element corresponds to a different system. Utilizing the HBM available on modern FPGAs, the full vectorized Thomas solver, which can be viewed as a single compute unit (CU), can be instantiated a number of times to obtain further parallel performance. Specific designs for applications with multiple CUs are discussed in Section 4. For a 3D application, the x- and y-dimension solves can be effectively pipelined, storing the resulting XY planes in on-chip memory without writing to external memory. However the z-dimension solve requires reading from external memory. As such 2D applications can be further optimized with unrolling. Again we discuss specific implementations with unrolling in Section 4.

### 3.2 Larger Systems Solves

Interleaved solving of systems requires on-chip memory proportional to the system size, $N$ and number of groups $g$. As such, the maximum size of the system that can be solved is limited by the FPGA on-chip memory resources. We can split the tridiagonal system into subsystems (or tiles) of size $M$ where each subsystem can be solved using a modified Thomas solver, where after a forward and backward phase, each unknown is expressed in terms of two unknowns $u_0$ and $u_{M-1}$: 

$$ a_iu_0 + u_i + c_ju_{M-1} = d_i, \quad i = 1, 2, ..., M-2 $$  

(6)

This results in a reduced tridiagonal system spread across each sub-domain as detailed by László et al. [13]). The unknowns at the beginning and end of each subsystem can be solved again using the Thomas algorithm, or indeed PCR. Finally, the result from the reduced system, is substituted back into the individual subsystems (see László et al. [13]) which implements a Thomas-PCR solver for GPUs.

The tiled-Thomas solver requires additional circuitry to solve the reduced system. To achieve higher performance, forward and backward phases over tiles can be interleaved. The reduced system size $N_R$ is double the number of tiles. Solving the reduced system with Thomas requires $2gN_r$ clock cycles. This should not exceed the clock cycles taken by the forward and backward phases over the tiles. At the end of the backward phase, results ($a^*, c^*$ and $d^*$ as noted in [13]) are stored in a FIFO buffer while the reduced system for each tile is computed. Then the reduced system results can be substituted back to complete the solve. Using a FIFO maintains the dataflow pipeline without stalling.
Figure 1: Datapath for x- and y-dim solves. 4-point data path width and 4× (vectorized) Thomas solvers.

Considering a system of size $N$, split into $t$ tiles (note then $N_t = 2t$), assume we interleave $g$ number of tiles using the Thomas-Thomas algorithm to solve a total number of $B$ systems. Then the total latency is given by (7):

$$\frac{2 + \lceil Bt/g \rceil}{N/t} g + \frac{g}{t} \times (2t) \times 2$$

(7)

The second term is for the reduced solve. The $g_r$ is similar to $g$, but it is equal to or larger than the number of interlaced systems for the reduced solve. It is $32$ for FP32 and $64$ for FP64 on the U280. Similarly, based on the latency for solving the first phase of the algorithm on a tile, the number of systems to be interlaced is $32/t$ for FP32 and $64/t$ for FP64. For larger $B$, we can see that the latency tends to $Bt/N/t$. Considering on-chip memory requirements the forward and backward phases of the modified Thomas can be shown to require $9 \times 2 \times g/t \times N$ words that can be satisfied by 9 memories setup as ping-pong buffers. Here we note that larger $t$ lead to lower memory requirement. The reduced solve requires much less memory, $7 \times 2 \times 2t \times \lceil g/t \rceil$ in the form of 7 ping pong buffers. Furthermore, a FIFO buffer would be required, of length equivalent to the maximum number of clock cycles spent on the reduced system, as we have to flush each point from the backward phase.

The reduced system solve can also be implemented with the PCR algorithm resulting in the latency given in (8):

$$\frac{2 + \lceil Bt/g \rceil}{N/t} g + \frac{(2t + I) \times \log(2t)}{t}$$

(8)

Again for larger $B$, this tends to $Bt/N/t$, however, there is a lower on-chip memory requirement of $(2t + I) \times \log(2t)$ words for each of the 3 FIFO buffers, due to the lower latency for reduced system solve in PCR. Since data flow design requires matching performance of solving tiles and the reduced system and as PCR is faster when solving reduced systems, the number of tiles can be increased even for smaller systems, further reducing the on-chip memory requirements for the first phase of the algorithm. As such we can expect the Thomas-PCR version to give better performance.

4 PERFORMANCE

In this section we examine the achieved performance for the above FPGA design strategy. First, we briefly compare the performance of our library to a current state-of-the-art FPGA tridiagonal solver library from Xilinx [4] which is based on PCR, demonstrating the higher performance gains from a batched Thomas-based solver as predicted by the analytical model developed in Section 3. Batching of systems is key to higher performance. Figure 2 presents the performance of 1D tridiagonal systems of size 128 and 1024, solved using the Xilinx library (xilinxlib-F1-128) compared to our Thomas algorithm-based library (tridsolvlib) and tiled-Thomas-PCR (Tiled-tridsolvlib) on a range of batch sizes. As demonstrated by the analytical model, for larger batch sizes the Xilinx library performed significantly slower than the Thomas based solver. Adding further optimizations, such as inner loop unrolling and a FIFO data path to the Xilinx solver (xilinxlib-F2) only marginally improved performance, leaving an order of a magnitude performance gap. We also observed that the PCR-based xilinxlib-F2 implementation consumed higher resources. Given that Tiled-tridsolvlib breaks the systems into 32 tiles, and gives faster solve times compared to tridsolvlib for small batch sizes due to smaller tiles being solved in an interleaved manner.

In the remainder of this section we focus on using our FPGA design strategy specifically, applied to representative, non-trivial applications. We investigate both 2D and 3D applications, with both FP32 and FP64 precision. Model predicted resource utilization estimates are used to determine initial design parameters and model predicted runtimes, which we compare to achieved runtimes on a Xilinx Alveo U280. We use Vivado C++ due to ease of use for configurations and support of some C++ constructs compared to OpenCL. However, OpenCL could equally be used to implement the same design. Finally, we compare performance on the FPGA to an NVIDIA Tesla V100 GPU using the tridiagonal solver library, tridsolver implemented by László et al. [13] [2] using its batched version presented by Reguly et al. [20]. This GPU library has been shown [7] to provide matching or better performance than the two current batch tridiagonal solver functions in Nvidia’s cuSPARSE library [6, 23] – cusparse<t>gtsvStridedBatch() and cusparse<t>gtsvInterleavedBatch(). Our experiments also confirmed these results for the applications evaluated in this paper. Additionally it features direct support for creating multi-dimensional solvers, whereas gtsvInterleavedBatch() requires data layout transformations, for example in between doing an x-solve and a y-solve to implement multi-dimensional problems while the cuSPARSE library variant is slower. Thus we use tridsolver in our evaluation throughout this paper, but note that cuSPARSE libs would have equally provided the same insights when compared to the FPGA solvers on the Xilinx U280. Given that previous work has demonstrated GPUs to provide significantly better performance than multi-threaded CPUs [13], we do not compare with CPU runs. Note that we only measure and present the time for the main iterative loop. As the applications carry out large numbers of iterations,
The first application is an Alternating Direction Implicit (ADI) based heat diffusion equation. The non-coalesced memory access pattern of the tridiagonal solver for each of two or three dimensions, calculating the RHS values using a 7-point stencil, followed by solving the tridiagonal system. The pipelining reduces the bandwidth requirement by half compared to the previous design. The first module, RHS + Tridslv(x-dim) + Tridslv(y-dim) and second module, Tridslv(z-dim), operate in parallel in a ping-pong fashion. This effectively increases the number of modules working in parallel to 24, considering the availability of HBM ports. The design now has a large pipeline start delay and is best utilized by batching large numbers of 3D meshes to obtain higher throughput.

The tridiagonal solver for each dimension is accumulated to $u$ before the next iteration. For the 3D ADI application, there are three calls to Tridslv. A GPU implementation has four kernels called by an iterative loop on the host. Fusing these kernels together does not improve performance as it requires global synchronization for data structure $d$ and the memory accesses are along different directions of the 3D mesh, leading to poor cache utilization. The non-coalesced memory access pattern of Tridslv(x-dim) is a challenge for GPUs. László et al. [13] improved performance through shared memory and register based transposing.

**Algorithm 3: 3D ADI Heat Application**

```plaintext
1: for $i = 0, i < niter, i++$ do
2:  Calculate RHS : $d = f_{xt}(u), a = \frac{1}{2}y, b = y, c = \frac{1}{2}y$
3:  Tridslv(x-dim), update $d$
4:  Tridslv(y-dim), update $d$
5:  Tridslv(z-dim), update $d$
6:  $u = u + d$
7: end for
```

The application consists of an iterative loop which starts by calculating the RHS values using a 7-point stencil, followed by calls to the tridiagonal solver for each of two or three dimensions, depending on the application. The updates from the tridiagonal solver, Tridslv, are accumulated to $u$ before the next iteration. For the 3D ADI application, there are three calls to Tridslv. A GPU implementation has four kernels called by an iterative loop on the host. Fusing these kernels together does not improve performance as it requires global synchronization for data structure $d$ and the memory accesses are along different directions of the 3D mesh, leading to poor cache utilization. The non-coalesced memory access pattern of Tridslv(x-dim) is a challenge for GPUs. László et al. [13] improved performance through shared memory and register based transposing.

**4.1 ADI Heat Diffusion Application**

The first application is an Alternating Direction Implicit (ADI) based heat diffusion equation. The high-level algorithm of the application in 3D is detailed in Alg. 3.

**Table 1: Experimental systems specifications.**

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA</td>
<td>Xilinx Alveo U280 [27]</td>
</tr>
<tr>
<td>DSP blocks</td>
<td>8490</td>
</tr>
<tr>
<td>BRAM / URAMs eM</td>
<td>(1487 blocks) / 34.5MB (960 blocks)</td>
</tr>
<tr>
<td>HBM</td>
<td>8GB, 460GB/s, 32 channels</td>
</tr>
<tr>
<td>DDR4</td>
<td>32GB, 38.4GB/s, in 2 banks</td>
</tr>
<tr>
<td>Host</td>
<td>AMD Ryzen Threadripper PRO 3975WX (32 cores)</td>
</tr>
<tr>
<td>HBM</td>
<td>32GB RAM, Ubuntu 18.04.6 LTS</td>
</tr>
<tr>
<td>Design SW</td>
<td>Vivado HLS, Vitis 2019.2</td>
</tr>
<tr>
<td>Run-Time</td>
<td>xrt_202020.2.9.317</td>
</tr>
<tr>
<td>GPU</td>
<td>Nvidia Tesla V100 PCIe [1]</td>
</tr>
<tr>
<td>Global Mem.</td>
<td>16GB HBM2, 900GB/s</td>
</tr>
<tr>
<td>Host Intel Xeon Gold 6252 @2.10GHz (48 cores)</td>
<td>256GB RAM, Ubuntu 18.04.3 LTS</td>
</tr>
<tr>
<td>HBM DDR4</td>
<td>32GB, 38.4GB/s, in 2 banks</td>
</tr>
<tr>
<td>HBM</td>
<td>8GB, 460GB/s, 32 channels</td>
</tr>
<tr>
<td>BRAM / URAMs</td>
<td>6.6MB (1487 blocks) / 34.5MB (960 blocks)</td>
</tr>
<tr>
<td>DSP blocks</td>
<td>8490</td>
</tr>
<tr>
<td>Host</td>
<td>AMD Ryzen Threadripper PRO 3975WX (32 cores)</td>
</tr>
<tr>
<td>DDR4</td>
<td>32GB RAM, Ubuntu 18.04.6 LTS</td>
</tr>
<tr>
<td>Design SW</td>
<td>Vivado HLS, Vitis 2019.2</td>
</tr>
<tr>
<td>Run-Time</td>
<td>xrt_202020.2.9.317</td>
</tr>
<tr>
<td>GPU</td>
<td>Nvidia Tesla V100 PCIe [1]</td>
</tr>
<tr>
<td>Global Mem.</td>
<td>16GB HBM2, 900GB/s</td>
</tr>
<tr>
<td>Host Intel Xeon Gold 6252 @2.10GHz (48 cores)</td>
<td>256GB RAM, Ubuntu 18.04.3 LTS</td>
</tr>
<tr>
<td>HBM DDR4</td>
<td>32GB, 38.4GB/s, in 2 banks</td>
</tr>
<tr>
<td>HBM</td>
<td>8GB, 460GB/s, 32 channels</td>
</tr>
<tr>
<td>BRAM / URAMs</td>
<td>6.6MB (1487 blocks) / 34.5MB (960 blocks)</td>
</tr>
<tr>
<td>DSP blocks</td>
<td>8490</td>
</tr>
<tr>
<td>Host</td>
<td>AMD Ryzen Threadripper PRO 3975WX (32 cores)</td>
</tr>
<tr>
<td>DDR4</td>
<td>32GB RAM, Ubuntu 18.04.6 LTS</td>
</tr>
<tr>
<td>Design SW</td>
<td>Vivado HLS, Vitis 2019.2</td>
</tr>
<tr>
<td>Run-Time</td>
<td>xrt_202020.2.9.317</td>
</tr>
<tr>
<td>GPU</td>
<td>Nvidia Tesla V100 PCIe [1]</td>
</tr>
<tr>
<td>Global Mem.</td>
<td>16GB HBM2, 900GB/s</td>
</tr>
<tr>
<td>Host Intel Xeon Gold 6252 @2.10GHz (48 cores)</td>
<td>256GB RAM, Ubuntu 18.04.3 LTS</td>
</tr>
<tr>
<td>HBM DDR4</td>
<td>32GB, 38.4GB/s, in 2 banks</td>
</tr>
<tr>
<td>HBM</td>
<td>8GB, 460GB/s, 32 channels</td>
</tr>
<tr>
<td>BRAM / URAMs</td>
<td>6.6MB (1487 blocks) / 34.5MB (960 blocks)</td>
</tr>
<tr>
<td>DSP blocks</td>
<td>8490</td>
</tr>
<tr>
<td>Host</td>
<td>AMD Ryzen Threadripper PRO 3975WX (32 cores)</td>
</tr>
<tr>
<td>DDR4</td>
<td>32GB RAM, Ubuntu 18.04.6 LTS</td>
</tr>
<tr>
<td>Design SW</td>
<td>Vivado HLS, Vitis 2019.2</td>
</tr>
<tr>
<td>Run-Time</td>
<td>xrt_202020.2.9.317</td>
</tr>
<tr>
<td>GPU</td>
<td>Nvidia Tesla V100 PCIe [1]</td>
</tr>
<tr>
<td>Global Mem.</td>
<td>16GB HBM2, 900GB/s</td>
</tr>
<tr>
<td>Host Intel Xeon Gold 6252 @2.10GHz (48 cores)</td>
<td>256GB RAM, Ubuntu 18.04.3 LTS</td>
</tr>
<tr>
<td>HBM DDR4</td>
<td>32GB, 38.4GB/s, in 2 banks</td>
</tr>
<tr>
<td>HBM</td>
<td>8GB, 460GB/s, 32 channels</td>
</tr>
<tr>
<td>BRAM / URAMs</td>
<td>6.6MB (1487 blocks) / 34.5MB (960 blocks)</td>
</tr>
<tr>
<td>DSP blocks</td>
<td>8490</td>
</tr>
<tr>
<td>Host</td>
<td>AMD Ryzen Threadripper PRO 3975WX (32 cores)</td>
</tr>
<tr>
<td>DDR4</td>
<td>32GB RAM, Ubuntu 18.04.6 LTS</td>
</tr>
<tr>
<td>Design SW</td>
<td>Vivado HLS, Vitis 2019.2</td>
</tr>
<tr>
<td>Run-Time</td>
<td>xrt_202020.2.9.317</td>
</tr>
<tr>
<td>GPU</td>
<td>Nvidia Tesla V100 PCIe [1]</td>
</tr>
<tr>
<td>Global Mem.</td>
<td>16GB HBM2, 900GB/s</td>
</tr>
<tr>
<td>Host Intel Xeon Gold 6252 @2.10GHz (48 cores)</td>
<td>256GB RAM, Ubuntu 18.04.3 LTS</td>
</tr>
<tr>
<td>HBM DDR4</td>
<td>32GB, 38.4GB/s, in 2 banks</td>
</tr>
<tr>
<td>HBM</td>
<td>8GB, 460GB/s, 32 channels</td>
</tr>
<tr>
<td>BRAM / URAMs</td>
<td>6.6MB (1487 blocks) / 34.5MB (960 blocks)</td>
</tr>
</tbody>
</table>

Figure 3: 2D ADI application datapath constructed from solver components.

An initial FPGA design implements the application as a single hardware unit given the data dependencies between the calls. This enables FPGA resource utilization to be maximized by implementing 6 CUs each having 8 Thomas solvers synthesized as a vectorized solver. The RHS calculation, which is a 3D explicit stencil loop was implemented using techniques similar to those in [12], as a separate module.

The intermediate results between CUs and RHS module were written/read to/from external memory. The number of CUs is then limited by the available HBM ports but not by any other resource. An improvement on this initial design fuses the generation of $a, b, c$ coefficients with the tridiagonal solver. This enables the required number of HBM ports to be reduced and synthesis of a maximum of 16 CUs. We opt for 12 CUs to reduce routing congestion which affects the maximum frequency achievable on the FPGA.

The x-dim and y-dim solves can be synthesized as separate modules, pipelining the X and Y dimension calculation without needing to buffer intermediate results in external memory. Essentially, XY planes are buffered in on-chip memory, but solvable mesh sizes are limited by BRAM/URAM usage. To also pipeline the z-dim solve the full mesh must be buffered on-chip which significantly limits the mesh size, hence we do not attempt it here. The pipelining reduces the bandwidth requirement by half compared to the previous design. The first module, RHS + Tridslv(x-dim) + Tridslv(y-dim) and second module, Tridslv(z-dim), operate in parallel in a ping-pong fashion. This effectively increases the number of modules working in parallel to 24, considering the availability of HBM ports. The design now has a large pipeline start delay and is best utilized by batching large numbers of 3D meshes to obtain higher throughput.
to start outputting the first result. Thus the full pipeline latency for the 3D ADI application is (9):
\[
L_{\text{adi,3D}} = n_{\text{iter}} \times \text{MAX}(L_{\text{rhs,xy}}, L_z)
\] (9)
\[
L_{\text{rhs,xy}} = (2xy/v) + (2xv/a + 3gx) + (2xy/a + 3gy) + \lfloor B/2NCU \rfloor (xyz/v)
\] (10)
\[
L_z = (2xz/a + 3gz) + \lfloor B/2NCU \rfloor (xyz/v)
\] (11)

Here, \(x, y\) and \(z\) are the sizes of the dimensions in each dimension, \(NCU\) is the number of CUs implemented on the FPGA and \(B\) is the total number of 3D meshes, i.e. the number of batches. The terms in (10) account for the 3D stencil computation in RHS, \(\text{Tridslv(x-dim)}\) including latency to transpose the \(x\)-lines, \(\text{Tridslv(y-dim)}\) including the reading/writing \(y\)-lines from the buffered \(x\)-lines, and the latency to process \(B\) meshes using \(NCU\) CUs respectively.

We take the maximum in (9) because the two modules need to be synchronized, as they swap their read and write locations after processing \(B/2\) meshes. The vectorization factor \(v\) is 8 for our design and \(g\) is 32 for FP32 and 64 for FP64. A minor consideration for obtaining improved predictions from the above model is when the number of points per clock cycle arriving to the vectorized solvers is different to \(a\) due to memory bandwidth. For example if we use a single HBM port to read two data structures and if we use a 256-bit data path, a lower number of points \(p\) will enter the datapath than \(a\). Then, replacing \(v\) by \(p\) is more accurate.

A similar design can be developed for the 2D ADI application, but now the functions in the iterative loop RHS, \(\text{Tridslv(x-dim)}\) and \(\text{Tridslv(y-dim)}\) all can be pipelined. This makes it possible to unroll the iterative loop by some factor \(f_j\). Note that the variable \(a\) is incremented each iteration (line 6 of Alg. 3), where the previous value of \(a\) needs to be input at the end of each unrolled iteration to carry out this increment. However \(RHS\) of each iteration also consume \(a\) and thus we use a delay-buffer (similar to ones used in StencilFlow [8]) implemented as an HBM FIFO to feed the previous values of \(a\) to the increment stage mentioned in line 6. Implementation of an HBM FIFO with data access dependency distance based on the data structures allocated on specific HBM banks makes global memory synchronization possible in the dataflow pipeline without additional HBM throughput cost. Unrolling the iterative loop reduces the total number of data structures in external memory. Hence we are able to assign dedicated ports for each data structure which enables better dataflow throughput. The performance model for the 2D application is given in (12).
\[
L_{\text{adi,2D}} = (n_{\text{iter}}/f_j) \times L_{\text{rhs,xy}}
\] (12)
\[
L_{\text{rhs,xy}} = f_j \times [(2x/v) + (2xv/a + 3gx) + (2xy/a + 3gy) + \lfloor B/2NCU \rfloor (xyz/v)]
\] (13)

Pipeline latency increases with the unroll factor \(f_j\), but for large \(B\) it results in a higher overall speedup. The size of the FIFO delay buffer is equivalent to the total delay of RHS, \(\text{Tridslv(x-dim)}\), and \(\text{Tridslv(y-dim)}\): \(2x/a + 2xv/a + 3gx + 3gy + 2xy/v\).

Figure 4 details the performance of the 2D ADI Heat diffusion application implemented in both FP32 (a) and FP64 (b) on the Alveo U280 and compares it to execution on the V100 GPU. The design parameters for each are noted in the graphs. Operating frequencies are 292MHz and 288MHz for FP32 and FP64 respectively. The design improved post Vivado implementation frequencies were possible due to multiple compute units with careful SLR placement and HBM bank assignment constraints, manually flattened loops with arbitrary wordlength counters and an optimally pipelined and vectorized design. In both FP32 and FP64 cases the coefficients \(a\), \(b\), and \(c\) are internally generated, on the FPGA. This means that only \(u\) is read. Performance results demonstrate the FPGA outperforming the GPU particularly for runs with large batch sizes. Additionally the predictive model accuracy is over 85% with large batched predictions being more accurate at over 90%. Inspecting the effective bandwidth on each device as detailed in the top two sub-tables in Table 2 provides insights into the superior performance of the FPGA. The bandwidth is computed by counting the total number of bytes transferred during the execution of each call in Alg. 3, looking at the mesh data accessed and dividing by the total time taken by each call. On the GPU, we have detailed the achieved bandwidth of the \(x\)- (Gx) and \(y\)-dim (Gy) solves. On the FPGA we show the full bandwidth achieved in the pipeline. The \(x\)-dim bandwidth on the GPU is significantly worse due to the \(8 \times 8\) transpose operations. Such lower bandwidth are also confirmed by László et al. [13]. We additionally confirmed the same performance when using cuSPARSE’s cusparse<t>gtsv2StridedBatch() library function for the \(x\)-solve. The higher performance of the FPGA can be attributed to the unrolling of the iterative loop, keeping intermediate results in fast on-chip memories, thus allowing higher bandwidth utilization for the data path and the internal generation of coefficients. The GPU tridiagonal solver library is not currently setup for such an optimization. Thus, the application writes \(a\), \(b\), and \(c\) to global memory after \(RHS\) and intermediate results also written/read between the two \(\text{Tridslv}\) calls whereas on the FPGA these stay on-chip. Even with modifications to the GPU library to generate coefficients internally which would improve GPU performance, we believe the FPGA results point to a very competitive solution.

<table>
<thead>
<tr>
<th>Mesh</th>
<th>(BW)-1500B</th>
<th>(E)-1500B</th>
<th>(BW)-3800B</th>
<th>(E)-3800B</th>
</tr>
</thead>
<tbody>
<tr>
<td>F Gx</td>
<td>Gy</td>
<td>G</td>
<td>F</td>
<td>Gx</td>
</tr>
<tr>
<td>32(^2)</td>
<td>350</td>
<td>134</td>
<td>418</td>
<td>1</td>
</tr>
<tr>
<td>64(^2)</td>
<td>524</td>
<td>184</td>
<td>528</td>
<td>3</td>
</tr>
<tr>
<td>128(^2)</td>
<td>692</td>
<td>204</td>
<td>517</td>
<td>34</td>
</tr>
</tbody>
</table>

Table 2: ADI Heat Diffusion App.: Achieved Bandwidth, \(BW\) (GB/s) and Energy, \(E\) (J)
Thomas solvers are selected to be V100. The FPGA on average consumed 75W while the GPU power sizes will be not processing very large meshes. If the tile sizes for the Thomas-RHS application for larger meshes can be modeled using (14):

$$\text{energy consumption of the FPGA is}$$

$$\sum v \cdot 8$$

A Thomas-Thomas based implementation for the 2D ADI-Heat application for larger meshes can be modeled using (14):

$$L_{\text{adi}, 2D, tiled} = \text{iter} \times (L_{\text{rhs}, x} + L_y)$$

$$L_{\text{rhs}, x} = 2x/v + 2ax/v + 3gx/t_1 + 4gt_1 + Bxy/v$$

$$L_y = 2yT_x + 3gty_1 + 4gt_2 + Bxy/v$$

In this case, RHS and x-solve can be pipelined but y-solve cannot as we are computing “tiles” along the x-dimension, a large amount of on-chip memory would be required to transpose the mesh. The explicit stencil computation in RHS does not require tiling as we are not processing very large meshes. If the tile sizes for the Thomas-Thomas solvers are selected to be $t_1$ and $t_2$ then the reduced system sizes will be $2t_1$ and $2t_2$. Equation (15) accounts for the latency for RHS with x-dimension solve where the terms correspond to

<table>
<thead>
<tr>
<th>Mesh</th>
<th>BW-60B</th>
<th>E-60B</th>
<th>BW-180B</th>
<th>E-180B</th>
</tr>
</thead>
<tbody>
<tr>
<td>256$^2$</td>
<td>206</td>
<td>117</td>
<td>238</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14</td>
<td>217</td>
<td>186</td>
</tr>
<tr>
<td>512$^2$</td>
<td>218</td>
<td>177</td>
<td>400</td>
<td>17</td>
</tr>
<tr>
<td>896$^2$</td>
<td>220</td>
<td>204</td>
<td>503</td>
<td>53</td>
</tr>
</tbody>
</table>

Figure 4(e) and (f) present the performance of the 2D ADI heat diffusion application on large meshes solved using Thomas-PCR and Thomas-Thomas hybrid implementations. Again we compare with the same mesh sizes solved on the GPU. Due to the RHS and Tridiagv(x-dim) being pipelined together, the FPGA achieves better HBM bandwidth utilization. The GPU also achieves good bandwidth utilization where it reaches bandwidth levels similar to batched smaller meshes (see Table 3 for BWs for Thomas-PCR, Thomas-Thomas gave similar results). The FPGA can be seen to be 2-3x more energy efficient than the GPU for the largest mesh sizes.

**4.2 Stochastic Local Volatility**

The second application we synthesize and evaluate comes from computational finance. It implements a stochastic local volatility (SLV) model, which describe asset price processes, particularly foreign
exchange rates [21]. A bached GPU implementation based on a 2nd order finite-difference scheme was developed for this problem using the ODS DSL by Reguly et al. [20]. It is a 2D application implemented in FP64 precision. Its high-level algorithm is detailed in Algo. 4. The
d calculation was aided by the Xilinx HLS tools where the exact datapath pipeline latency was estimated to obtain buffer sizes adequate for an implementation.

The motivation for batched solves of multi-dimensional tridiagonal systems primarily comes from financial computing where, for example, computing prices of financial options and managing risk by hedging options leads to the need to solve Algo. 4 type applications with different sets of coefficients [20]. Additionally carrying out extensive speculative scenarios required by regulators under various market conditions to evaluate a bank’s exposure means that there are large numbers of options in the order of thousands to hundreds of thousands to be computed every day. Such workloads would entail large numbers of roughly identical PDE problems to be solved which are well suited to be batched together.

Figure 5 and Table 4 details the runtime, bandwidth, and energy performance of the SLV application implementation. Only two specific mesh sizes were available from the authors of the original code [20], each was batched up to 3000 batches of 2D meshes for this evaluation. The application is significantly more complex given the additional explicit stencil loops as well as the tridiagonal solvers. The runtimes here were obtained with the FPGA operating at 253.5MHz. As can be seen from the figures, the FPGA in some cases is faster than the V100 GPU, but for the largest batch sizes we attempted here, it is 8%-70% slower than the GPU. However the FPGA solution is over 30% more energy efficient for large batch solves over the GPU. The achieved bandwidth on the FPGA is approximately at the same level as the 2D ADI FP64 version. Runtime predictions from the model were also observed to be over 90% accurate for all cases.

5 RELATED WORK

Earlier works implementing tridiagonal system solvers on FPGAs such as by Oliveira et al. [17], Warne et al. [25] and Zhang et al. [28] used low-level Hardware Description languages (HDL) such as VHDL or Verilog for implementing the Thomas algorithm. These designs were restricted to solving 1D or 1D batched tridiagonal systems, instead of full multi-dimensional applications. Later, HLS tools-based solutions include [14–16, 26]. The Xilinx library also implements a Douglas ADI solver [10] a multi-dimensional solver based on their PCR based solver [4].

In comparison, the HLS-based synthesis presented in this paper, targets the solution of multiple tridiagonal systems and in multiple dimensions as commonly found in real-world applications. It uses the Thomas algorithm demonstrating that together with techniques such as batching of systems [12], it provides higher throughput for small and medium sized systems. The Thomas algorithm requires a relatively smaller amount of DSP resources than the more computationally intensive PCR algorithm. For larger systems that do not directly fit in a single FPGA, we develop novel Thomas–Thomas and

<table>
<thead>
<tr>
<th>Batch</th>
<th>TBW (GB/s)</th>
<th>GxBW (GB/s)</th>
<th>GyBW (GB/s)</th>
<th>F(E)</th>
<th>G(E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>55.24</td>
<td>3.04</td>
<td>28.01</td>
<td>0.13</td>
<td>0.45</td>
</tr>
<tr>
<td>300</td>
<td>202.31</td>
<td>16.48</td>
<td>176.51</td>
<td>0.35</td>
<td>1.02</td>
</tr>
<tr>
<td>3000</td>
<td>281.06</td>
<td>123.84</td>
<td>327.65</td>
<td>2.51</td>
<td>4.75</td>
</tr>
<tr>
<td>1000×50 mesh</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>124.63</td>
<td>51.28</td>
<td>109.65</td>
<td>3.98</td>
<td>3.76</td>
</tr>
<tr>
<td>300</td>
<td>278.87</td>
<td>235.22</td>
<td>238.34</td>
<td>17.79</td>
<td>22.26</td>
</tr>
<tr>
<td>3000</td>
<td>318.36</td>
<td>421.77</td>
<td>429.21</td>
<td>155.82</td>
<td>216.40</td>
</tr>
</tbody>
</table>
Thomas-PCR solvers to handle a number of partitioned systems and then a reduced system solve so that it can operate with the available limited on-chip memory of a single FPGA. Our designs also use High Bandwidth Memory (HBM) on modern FPGAs which helps to scale the design to multiple compute units. To our knowledge, the 2D/3D ADI and SLV applications developed in this work, motivated by real-world implicit problems on FPGAs is also novel; SLV being one of the few non-trivial applications using multi-dimensional tridiagonal solvers presented in literature. The Thomas based solver developed in this paper gives higher performance than the current PCR based Xilinx library, as we showed in Section 4. Additionally, the predictive analytic model and the performance comparison with a state-of-the-art GPU based tridiagonal solver library gives a much needed frame of reference for evaluating our FPGA design’s performance, providing insights into the feasibility and profitability of an FPGA design for realistic workloads.

6 DISCUSSION

The experiments in Section 4.1 show better performance on the Xilinx Alveo U280 compared to the Nvidia V100 for ADI 2D and ADI 3D applications in both FP32 and FP64 formats. Key optimizations possible on the FPGA, such as pipelining and fusing coefficient generation with tridiagonal solvers leads to this performance gain. These optimizations helped to achieve higher effective bandwidth on FPGA although U280 HBM’s theoretical bandwidth is 460GB/s and the V100 HBM’s bandwidth is 900 GB/s. Additionally, lower FPGA resource consumption due to these optimization makes it possible to scale to multiple compute units on the Alveo U280. Implementation of an 8x8 transpose on the FPGA enabled higher throughput for Tridslv(x – dim) even with non-coalesced memory accesses, while the GPU implementation using shared memory based transpose and Tridslv to address non-coalesced accesses suffers significant performance loss. In Section 4.2, the FPGA demonstrates competitive performance with the GPU for the SLV application. However, the computationally intensive complex coefficient calculation using 10-point stencils makes it hard to fuse with the Thomas solver (another Thomas solver library variant is used here) and results in higher FPGA resource usage, limiting the number of implementable compute units. Due to this, the GPU performs better than the FPGA for SLV applications on larger meshes. Future FPGAs with more DSP units/ floating point primitives will provide better performance than the Alveo U280. However, SLV with smaller meshes/batches is better matched to the FPGA due to the low latency FPGA data movement as well as lower kernel call overhead as the iterative loop is implemented within the FPGA kernel.

7 CONCLUSION

We developed a new FPGA-based tridiagonal solver library aimed at solving multiple multi-dimension tridiagonal systems on FPGAs. Key new features of the library include dataflow techniques and optimizations for gaining high throughput, through batching multiple system solves, replication of compute units, and utilization of High Bandwidth Memory on modern FPGAs. The Thomas algorithm was shown to be effective, even with its loop carried dependencies, due to its simplicity and lower resource consumption. This somewhat subverts the conventional expectation of the more parallel PCR or spike algorithms being the best suited for high performance on parallel architectures. Our library significantly outperforms the Xilinx tridiagonal library that uses the PCR algorithm, for larger batch sizes. However, for larger mesh sizes a hybrid Thomas-PCR or Thomas-Thomas solution was required to overcome the limitations of on-chip memory and demonstrated credible performance overall with batched configurations.

Two representative applications (1) a heat diffusion problem based on the ADI method and (2) a stochastic local volatility (SLV) model from the financial computing domain that rely on the solution of multi-dimensional tridiagonal systems were implemented using the new library on a Xilinx Alveo U280 FPGA. As part of the design process a predictive analytic model that estimates the runtime performance of FPGA designs was also developed to assist in design space evaluations. The FPGA performance was compared to optimized solutions of the same applications on a modern Nvidia Tesla V100 GPU, showing competitive performance, sometimes even surpassing the performance on the GPU. This was due to designs creating longer pipelines keeping intermediate results on fast FPGA on-chip memory.

Even when runtime is inferior to the GPU, significant energy savings, over 30% for the most complex application (SLV) with large batch sizes, were observed. Considering the motivating real-world scenario for such an application from the financial computing domain, such energy savings point to a significant cost benefit in overall operation. The predictive model provides over 85% accuracy illustrating its significant utility in developing profitable FPGA designs. The results showcase a key class of applications and their characteristics where the FPGA is able to provide competitive performance on-par with GPUs, with the added benefit of large energy savings. The techniques and optimizations required to achieve high
The FPGA library, the 2D/3D ADI heat diffusion application and optimized GPU source code developed in this work are available as open source software at [5], including additional results from a Xilinx Alveo U50 FPGA, which support the Alveo U280 results. Future work will explore the use of FPGA hardware from Intel, the other major FPGA device vendor.

ACKNOWLEDGMENTS

Gihan Mudalige was supported by the Royal Society Industry Fellowship Scheme (INF/R1/1800 12). István Reguly was supported by National Research, Development and Innovation Fund of Hungary (PD 124905), under the PD_17 funding scheme. We are grateful to Xilinx for their hardware and software donations and Jacques Du Toit and Tim Schmielau at NAG UK for their advice and making the SLV application available for this work.

REFERENCES