High Throughput Multidimensional Tridiagonal System Solvers on FPGAs

Kamalavasan Kamalakkannan
Gihan R. Mudalige
University of Warwick, UK
[kamalavasan.kamalakkannan, g.mudalige]@warwick.ac.uk

Istvan Z. Reguly
Pazmany Peter Catholic University
Hungary
reguly.istvan@itk.ppke.hu

Suhaib A. Fahmy
King Abdullah University of Science and Technology (KAUST)
Saudi Arabia
suhaib.fahmy@kaust.edu.sa

ABSTRACT

We present a high performance tridiagonal solver library for Xilinx FPGAs optimized for multiple multi-dimensional systems common in real-world applications. An analytical performance model is developed and used to explore the design space and obtain rapid performance estimates that are over 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 over 30%. Through this design, we learn lessons about the types of applications where FPGAs can challenge the current dominance of GPUs.

CCS CONCEPTS

• **Computer systems organization** → **Reconfigurable computing**; *Multicore architectures*; • **Mathematics of computing** → **Mathematical software performance**.

KEYWORDS

High performance computing, field programmable gate arrays, tridiagonal solvers.

ACM Reference Format:
Kamalavasan Kamalakkannan, Gihan R. Mudalige, Istvan Z. Reguly, and Suhaib A. Fahmy. 2022. High Throughput Multidimensional Tridiagonal System Solvers on FPGAs. In *2022 International Conference on Supercomputing (ICS '22), June 28–30, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524059.3532371

1 INTRODUCTION

Tridiagonal systems of equations are solved in a wide range of High Performance Computing (HPC) applications, particularly as part of the numerical approximation of multi-dimensional partial differential equations (PDEs). In computational finance, the frequently used Alternating Direction Implicit (ADI) time discretization problem (see Paceman and Rachford [20], and Douglas and Gunn [10]) requires multiple tridiagonal systems of equations to be solved in each dimension. The large number of independent tridiagonal systems, often in multiple dimensions, offer significant parallelization opportunities on modern multi-core and many-core architectures. Recent work such as László et al. [15] has demonstrated significant speedups, re-evaluating the well known tridiagonal solver algorithms, Thomas [26], PCR [11], and combinations of the two.

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 launched in an iteration loop on the host CPU, hence requiring kernel input and output data to be moved through GPU global memory repeatedly. 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 workload 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 [19, 29, 32] and high-level synthesis tools [4, 16–18, 30], 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 is also limited in current literature, creating a need for 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 (Section 2), 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 (Section 3). The architecture exploits High Bandwidth Memory (HBM) to combine multiple dimension solves and explicit loops, along with batched execution of multiple independent solves. We present the optimized design of two non-trivial applications, a 2D and 3D ADI heat diffusion solver, implemented with both single precision (FP32) and double precision (FP64) floating point representations, and a

ICS '22, June 28–30, 2022, Virtual Event, USA

2D Stochastic-Local Volatility (SLV) model application from the financial computing domain.

- An analytical performance model is developed (Section 3) 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 runtime 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 FPGA compared to an HPC-grade Nvidia V100 GPU, for both FP32 and FP64 precision, and much improved energy efficiency (Sections 4 and 5).

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 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 FPGA platforms.

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).

$$a_i u_{i-1} + b_i u_i + c_i u_{i+1} = d_i, \quad i = 0, 1, ..., N-1 \tag{1}$$

$$\begin{bmatrix} b_0 & c_0 & 0 & \dots & 0 \\ a_1 & b_1 & c_1 & \dots & 0 \\ 0 & a_2 & b_2 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{N-1} & b_{N-1} \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{N-1} \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{N-1} \end{bmatrix} \tag{2}$$

The solution to such systems of equations is well known. The Thomas algorithm [26] (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 $O(N)$. In contrast, the PCR algorithm [11](see Algo. 2), operates on a normalized matrix so that $b_i = 1$ and then for each matrix row i, subtracts multiples of rows $i \pm 2^0, 2^1, 2^2, ..., 2^{P-1}$, where P is the smallest integer such that $2^P \geq N$. This leads to each iteration reducing each of the current systems into two systems of half the size. After P steps, all of the modified a and c coefficients are zero, leaving values for the unknowns u_i. In PCR, the iterations of the inner loop do not depend on each other, which is well suited for traditional multi-core/many-core architectures such as CPUs and GPUs allowing multiple threads to be used to solve each tridiagonal system. However, PCR has a complexity of $O(N \log N)$ and is more computationally expensive than the Thomas algorithm, which for an FPGA implementation poses an important consideration, (as examined in Section 3) due to the limited availability of resources.

Algorithm 1: `thomas`(a, b, c, d, u)

$d_0^* \leftarrow d_0 / b_0$
$c_0^* \leftarrow c_0 / b_0$
for $i = 1, 2, ..., N-1$ **do**
  $r \leftarrow 1/(b_i - a_i c_{i-1}^*)$
  $d_i^* \leftarrow r(d_i - a_i d_{i-1}^*)$
  $c_i^* \leftarrow r c_i$
end for
$u_{N-1} \leftarrow d_{N-1}$
for $i = N-2, ..., 1, 0$ **do**
  $u_i \leftarrow d_i^* - c_i^* u_{i+1}$
end for
return u

Algorithm 2: `pcr`(a, b, c, d, u)

for $p = 1, 2, ..., P$ **do**
  $s \leftarrow 2^{p-1}$
  for $i = 0, 1, ..., N-1$ **do**
    $r \leftarrow 1/(1 - a_i^{(p-1)} c_{i-s}^{(p-1)} - c_i^{(p-1)} a_{i+s}^{(p-1)})$
    $a_i^{(p)} \leftarrow -r(a_i^{(p-1)} a_{i-s}^{(p-1)})$
    $c_i^{(p)} \leftarrow -r(c_i^{(p-1)} c_{i+s}^{(p-1)})$
    $d_i^{(p)} \leftarrow r(d_i^{(p-1)} - a_i^{(p-1)} d_{i-s}^{(p-1)} - c_i^{(p-1)} d_{i+s}^{(p-1)})$
  end for
end for
$u \leftarrow d^{(P)}$
return u

The SPIKE algorithm [21] 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 $A_1, ..., A_p$ and S is the so called spike matrix. The solution to the system then becomes, $DSx = d$ 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_i, each $A_i Y_i = d_i$ can be solved independently. Solving $Sx = Y$ requires only solving a reduced penta-diagonal system (see Wang et al. [28] 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 axes. For example a 3D system with $N_x \times N_y \times N_z$ mesh points will have $N_y \times N_z$ tridiagonal systems when solving along the first dimension (each system with size N_x), $N_x \times N_z$ along the second (each with size N_y) and so on. The ADI method, (used in the applications in this work), repeatedly solves 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, 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 an issue on traditional architectures such as CPUs or GPUs [15]. An FPGA design must also carefully consider memory performance when solving such multi-dimensional applications.

3 FPGA ARCHITECTURE DESIGN

FPGAs can offer high performance computation 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 for interconnecting and control logic, fast on-chip block memories (BRAM/URAM) of kilobit capacities for buffering, clock modules for managing different data rates, and a rich routing fabric to connect these elements into a large logical architecture. Optimizing the datapaths around the capabilities of the low level primitives, especially DSP blocks, allows for high operating frequency to be achieved, but requires considered application of pipelining [23]. 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 typically includes large capacity external DRAM (32 GB on the U280) and potentially High Bandwidth Memory (HBM). Managing the movement of data between these memories and the internal FPGA 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 must 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 through fast on-chip memory. For an 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 System 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_i^* and c_i^*, each iteration of the forward path loop must be executed serially, incurring the full arithmetic pipeline latency, l_f (≈30 clock cycles on a Xilinx U280 FPGA for FP32), to complete the forward loop datapath. 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 subtractions, 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 $(N+l) \times logN$. Here we assume that the outer loop is executed serially and a fully pipelined inner loop, i.e., an initiation interval of one. Given inner loop iterations are independent, they can be unrolled by some factor $f_U = 2, 3, \ldots$ which will then require $f_U\times$ the resources to implement the inner loop. The total clock cycles consumed will then be $(N/f_U+l)\times logN$. 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 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 l_f. 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 total tridiagonal systems to be solved, i.e. a batch size of B, then the total latency with Thomas is given by (3):

$$(3 + \lceil B/g \rceil) \times gN \tag{3}$$

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

$$(BN/f_U + l) \times logN \tag{4}$$

For large B, dividing (4) by (3) gives a factor of $logN/f_U$ pointing to the fact that the batched Thomas solver is $logN$ times faster than batched PCR, for $f_U = 1$. Thus, to match the Thomas solver latency, a batched PCR implementation needs an unroll factor $f_U = logN$. 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 amount of FPGA resources. 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):

$$(3 + \lceil Bm/g \rceil) \times gN/m + mC + 3 \times gN/m \tag{5}$$

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 clock cycles spent on the pentadiagonal reduced system solve is BmC (assuming a linear latency model), only the latency for first stage of the pentadiagonal solver is added to equation 5 as all three modules are pipelined. The final term is the added delay due to back-substitution stage which is again a Thomas solver. 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 $BmC \geqslant BN$ then dataflow must stall for some time, decreasing throughput. Resource consumption of the LU/UL factorizations requires 3× 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 c_{i-1}, d_{i-1} and u_{i+1} values needs to be stored in on-chip memory such that they can be used in subsequent (i^{th}) 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 must 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. The latencies for writing to the ping-pong buffer, firstly for a, b, c, d belonging to the first group of systems, then writing the resulting c^*, d^* in forward solve and finally writing u in backward solve, contribute to the latency term $3gN$ in (3). Here we assume, inputs a, b, c, d come from FIFO and output u is written back to FIFO. If inputs/outputs are read/written to on-chip memory instead, then (3) becomes $(1 + \lceil B/g \rceil) \times gN$.

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, b, c, d, c^*, d^* and u vectors, where each consumes $2gN$ words in the ping-pong buffers. The total $14gN$ requirement with dual port memory can be satisfied with 7× dual port block RAMs (URAM/BRAM) each with a capacity of $2gN$. Additionally there is a need to store g values of the $(i-1)^{th}$ iteration separately, requiring 3 on-chip memories with a capacity of g words.

Data transfer from external memory to on-chip memory plays a crucial role 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) of 256 in all three dimensions, then a solve along the x-dimension will have YZ (256×256 in this case) systems to be solved, each corresponding to an *x-line* 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 must 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 x-lines can be set up to be solved in groups (g) of 32. Here, the 1st Thomas solver datapath 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 dataflow 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 8×8 transpose to feed that to 8 different solvers (see Figure 1(a) for an illustration of the issue with a 4×4 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 System 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}:

$$a_i u_0 + u_i + c_i u_{M-1} = d_i, \quad i = 1, 2, ..., M-2 \tag{6}$$

This results in a reduced tridiagonal system spread across each sub-domain as detailed by László et al. [15]). 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. [15] which implements a Thomas-PCR solver for GPUs).

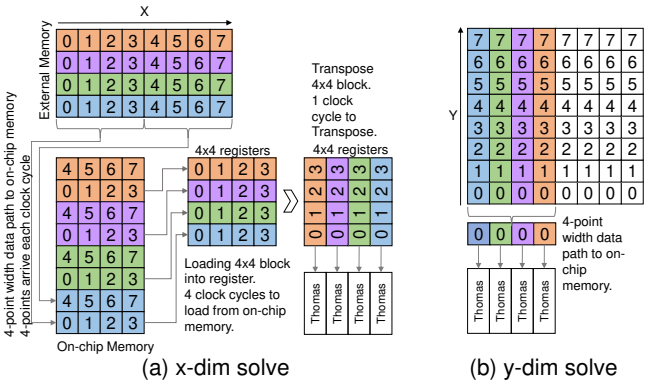


Figure 1: Datapath for x- and y-dim solves. 4-point data path width and 4× (vectorized) Thomas solvers.

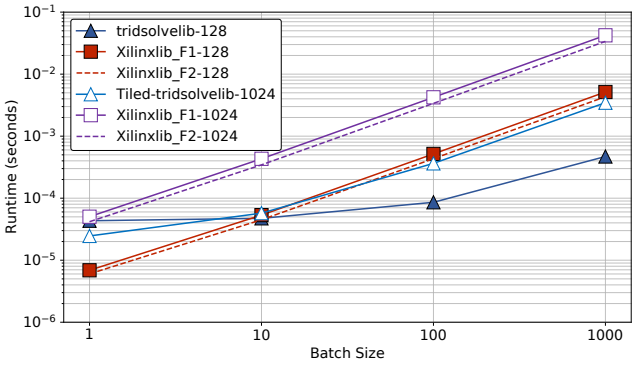


Figure 2: Proposed `tridsolvlib` vs `xilinxlib` (FP32) performance for system sizes of 128 and 1024.

The *tiled*-Thomas-Thomas solver requires additional computation 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 [15]) 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.

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

$$(3 + \lceil Bt/g \rceil) \times \lceil N/t \rceil g + g_r \times (2t) \times 2 \tag{7}$$

The second term is for the reduced solve. The g_r is similar to g, but it is equal to or larger than number of interleaved 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 interleaved is $\lceil 32/t \rceil$ for FP32 and $\lceil 64/t \rceil$ for FP64. For larger B, we can see that the latency tends to $Bt\lceil N/t \rceil$. 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 on-chip 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 solved tiles from the backward phase.

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

$$(3 + \lceil Bt/g \rceil) \times \lceil N/t \rceil g + (2t + l) \times log(2t) \tag{8}$$

Again for larger B, this tends to $Bt\lceil N/t \rceil$, however, there is a lower on-chip memory requirement of $(2t + l) \times log(2t)$ words for each of the 3 FIFO buffers, due to the lower latency for reduced system solve in PCR. Since dataflow 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 EVALUATION

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 performance 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, in FP32, solved using the Xilinx library (`xilinxlib-F1`) compared to our Thomas algorithm-based library (`tridsolvlib`) and tiled Thomas-PCR (`Tiled-tridsolvlib`) on a range of batch sizes. As predicted by the 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 improves performance, leaving an order of a magnitude performance gap. We also observe that the PCR-based `xilinxlib-F2` implementation consumes higher resources. `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. The performance models are used to determine initial design parameters and 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. Resources are estimated, with

Table 1: Experimental systems specifications.

FPGA	Xilinx Alveo U280 [31]
DSP blocks	8490
BRAM/URAM	6.6MB (1487 blocks)/34.5MB (960 blocks)
HBM	8GB, 460GB/s, 32 channels
DDR4	32GB, 38.4GB/s, in 2 banks
Host	AMD Ryzen Threadripper PRO 3975WX (32 cores)
	512GB RAM, Ubuntu 18.04.6 LTS
Design SW	Xilinx Vivado HLS, Vitis 2019.2
Run-Time	Xilinx XRT 202020.2.9.317
GPU	Nvidia Tesla V100 PCIe [1]
Global Mem.	16GB HBM2, 900GB/s
Host	Intel Xeon Gold 6252 @2.10GHz (48 cores)
	256GB RAM, Ubuntu 18.04.3 LTS
Compilers, OS	nvcc CUDA 10.0.130, Debian 9.11

the aid of Vivado HLS tools. Finally, we compare performance on the FPGA to an Nvidia Tesla V100 GPU using the tridiagonal solver library, `tridsolver` implemented by László et al. [2, 15] using its batched version presented by Reguly et al. [22]. 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, 27] – `cusparse<t>gtsv2StridedBatch()` 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. The cuSPARSE `gtsv2StridedBatch()` library variant was observed to be 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 [15], we do not compare with CPU implementations. Note that we only measure and present the time for the main iterative loop. As the applications carry out large numbers of iterations, the data copied to the device (on both devices via PCIe, incurring similar overheads), is used repeatedly. Therefore, the transfer overhead is amortized. Furthermore, with large multi-batch execution in real workloads, the initial transfer is further hidden behind computation. Hence, data copy time from host to device (both on FPGA and GPU) are not included in our results.

Table 1 briefly details the specifications of the FPGA and the GPU systems (both hardware and software) used in our evaluation. The Nvidia V100 is based on 12nm technology while the Xilinx U280 is 16nm. The GPU also has a memory bandwidth of 900GB/s, nearly twice that of the U280's 460GB/s. Thus we selected the V100 as a fair but challenging competitor.

4.1 ADI Heat Diffusion Application

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

Algorithm 3: `3D ADI Heat Application`

1: **for** $i = 0, i < n_{iter}, i++$ **do**
2: `Calculate RHS` : $d = f_{7pt}(u), a = \frac{-1}{2}\gamma, b = \gamma, c = \frac{-1}{2}\gamma$
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`s 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. [15] improved performance through shared memory and register based transposing.

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 [13], 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. Xilinx dataflow design synthesis requires separate data structures for independent read and write operations. We introduce two data

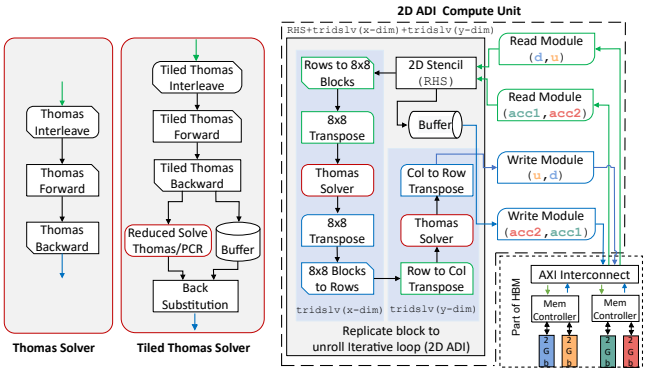


Figure 3: 2D ADI application datapath constructed from solver components.

structures for accumulation in line 6 of Algo 3. But due to limited HBM ports, we must share a single HBM port between two data structures. This limits the dataflow per data structure from/to the HBM ports as well as the size of data structure, given a single HBM bank has a capacity of 256MB. This final design gave the best performance in our evaluations.

The component model in (3) can be combined with the delays due to buffering (ping-pong buffers for the 8×8 transpose, row-to-col, rows-to-8×8-block data flow and window buffers for stencil computations) to obtain an application performance model. These delays are determined by the clock cycles needed to fill the buffers in order to start outputting the first result. Thus the full pipeline latency for the 3D ADI application is (9):

$$L_{adi,3D} = n_{iter} \times MAX(L_{rhs+xy}, L_z) \tag{9}$$

$$L_{rhs+xy} = (xy/v) + (2vx/v + 3gx) + (2xy/v + 3gy) + \lceil B/2N_{CU} \rceil (xyz/v) \tag{10}$$

$$L_z = (2xz/v + 3gz) + \lceil B/2N_{CU} \rceil (xyz/v) \tag{11}$$

Here, x, y and z are the sizes of systems in each dimension, N_{CU} 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, `Tridslv(x-dim)` including latency to transpose the x-lines, `Tridslv(y-dim)` including the reading/writing y-lines from the buffered x-lines, and the latency to process B meshes using N_{CU} 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 v 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 v. 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, `Tridslv(x-dim)` and `Tridslv(y-dim)` can all be pipelined. This makes it possible to unroll the iterative loop by some factor f_U. Note that the variable u is incremented each iteration (line 6 of Algo. 3), where the previous value of u must be input at the end of each unrolled iteration to carry out this increment. However the RHS of each iteration also consumes u 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 u to the increment stage on line 6. Implementation of an HBM FIFO with a 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 overall structure of the 2D design, combining component modules is illustrated in Figure 3. A similar illustration can be conceived for the 3D ADI application, which we do not show here. The performance model for the 2D application is given in (12).

$$L_{adi,2D} = (n_{iter}/f_U) \times L_{rhs+xy} \tag{12}$$

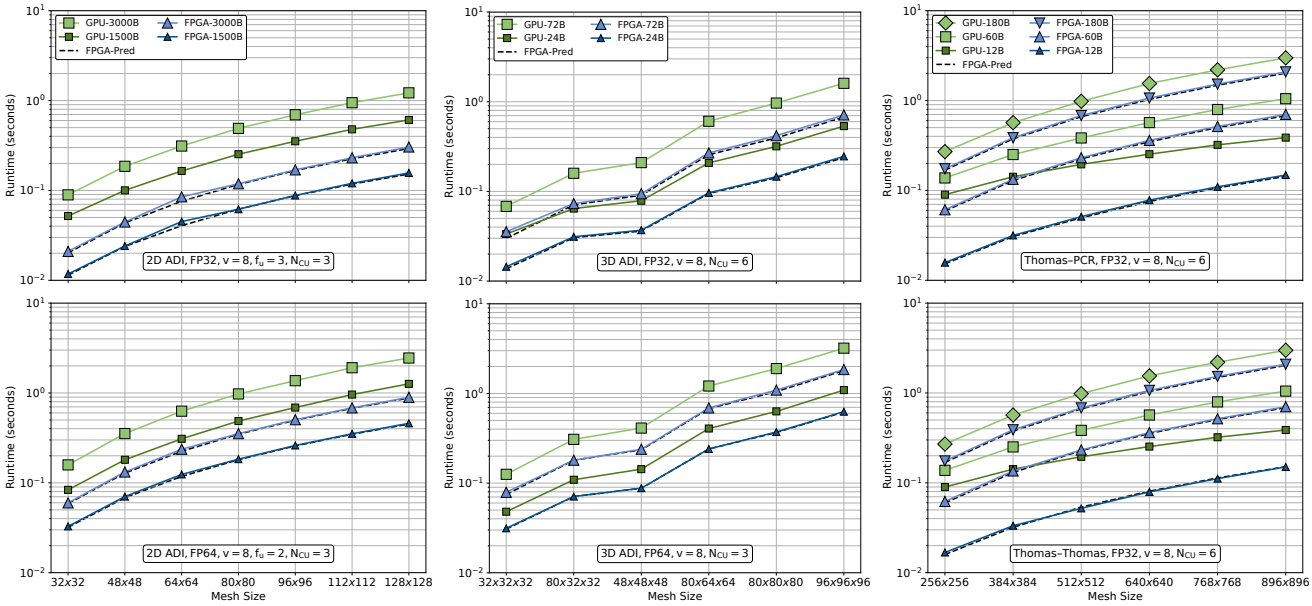
$$L_{rhs+xy} = f_U \times [(x/v) + (2vx/v + 3gx) + (2xy/v + 3gy)] + \lceil B/N_{CU} \rceil (xy/v) \tag{13}$$

Pipeline latency increases with the unroll factor f_U, 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, `Tridslv(x-dim)`, and `Tridslv(y-dim)` : $x/v + 2vx/v + 3gx + 3gy + 2xy/v$.

Figure 4 (a) details the performance of the 2D ADI Heat diffusion application implemented in both FP32 and FP64 on the FPGA and compares it to execution on the GPU. The design parameters for each are noted in the graphs. Operating frequencies are 292MHz and 288MHz for FP32 and FP64 respectively. These improved post implementation frequencies were possible due to multiple compute units with careful SLR placement and HBM bank assignment constraints, manually flattened loops with arbitrary word length 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.

We see that the performance model accuracy is over 85% with large batched predictions being more accurate at over 90%. The prediction errors in the models are due to omitting a number of minor latencies for simplicity. While the models accounts for only the latency in loops, real synthesized circuit on the FPGA will have additional stages to complete before a loop. These include calculating the loop invariants and initial values and state transition to function calls. For modeling loops, we do not account for the hardware pipeline latency - i.e. the clock cycles required between FIFO read and write and arithmetic hardware pipeline. However, these can be obtained from the HLS kernel schedule viewer to refine and improve predictions. We also do not account for latency incurred on the first external memory transfer. All of the above latencies are less than a few hundred clock cycles and become insignificant when considering total runtimes of larger mesh or batch sizes as can be seen from the above results.

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



(a) 2D ADI: 120 iter. **(b) 3D ADI: 100 iter.** **(c) 2D ADI-Tiled: 100 iter.**

Figure 4: ADI Heat Diffusion application performance.

the total number of bytes transferred during the execution of each call in Alg. 3, looking at the mesh data accessed and dividing it 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 block transpose operations. Such lower bandwidths are also confirmed by László et al. [15]. 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 does not support internal coefficient generation. Thus, the application writes a, b, c and u to global memory after RHS and intermediate results also written/read between the two `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, particularly when batching large meshes that can fit within the resource constraints of the FPGA, for this application.

The first two sub-tables in Table 2 also detail the energy consumption of the 2D runs. The `xbutil` utility was used to measure power during FPGA execution, while `nvidia-smi` was used for the GPU. The FPGA on average consumed 75W while the GPU power draw ranged from 50W to 250W. Results indicate that the FPGA energy consumption is approximately 5–6× lower for this 2D problem.

Figure 4 (b) and the bottom two sub-tables in Table 2 detail the performance of the 3D ADI heat diffusion application in FP32 and FP64 respectively. Again we see performance trends similar to the 2D case, however we were only able to run smaller batch sizes due to HBM memory limitations for 3D meshes. On the GPU, again, apart from the x-dim solve, we observe good achieved bandwidth. On the FPGA the achieved bandwidth is poorer due to no unrolling of the iterative loop as done in the 2D case, where there are 3 CUs each unrolled by a factor of 3. The sharing of HBM ports as described in the design of this application limits the data flow per data structure further reducing achieved bandwidth. The energy consumption of the FPGA is 3–4× lower than the GPU's.

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

$$L_{adi,2D,tiled} = n_{iter} \times (L_{rhs+x} + L_y) \tag{14}$$

$$L_{rhs+x} = x/v + 2vx/v + 3gx/t_1 + 4gt_1 + Bxy/v \tag{15}$$

$$L_y = 2yT_x/v + 3gy/t_2 + 4gt_2 + Bxy/v \tag{16}$$

In this case, RHS and x-solve can be pipelined but y-solve cannot as we are computing "tiles" along the y-dim lines, 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 the latencies of the stencil, the data path, modified Thomas solve, and the reduced solve. Similarly (16) gives the y-dimension solve latency. Note that here we have used T_x (this is different to $t1$) as the tile size for the y-dim data path where we buffer $T_x \times y$ sized planes. Note also that we have selected the number of interleaved systems and interleaved reduced systems to be equal (i.e. $g = g_r$

Table 2: ADI Heat Diffusion App. : Achieved Bandwidth (GB/s) and Energy (J) on the FPGA (F) and GPU(G).

2D FP32 (120 iterations, $f_U = 3$)										
Batch Size	1500					3000				
	Bandwidth			Energy		Bandwidth			Energy	
Mesh	F	Gx	Gy	F	G	F	Gx	Gy	F	G
32^2	501	134	418	1	7	563	164	493	2	13
64^2	524	184	528	3	23	556	199	553	6	42
128^2	602	207	563	12	86	620	206	565	23	168
2D FP64 (120 iterations, $f_U = 2$)										
Batch Size	1500					3000				
	Bandwidth			Energy		Bandwidth			Energy	
Mesh	F	Gx	Gy	F	G	F	Gx	Gy	F	G
32^2	360	184	508	2	10	395	196	543	4	21
64^2	380	203	557	9	42	399	203	529	18	88
128^2	411	204	517	34	179	422	210	551	67	355
3D FP32 (100 iterations)												
Batch Size	24						72					
	Bandwidth				Energy		Bandwidth				Energy	
Mesh	F	Gx	Gy	Gz	F	G	F	Gx	Gy	Gz	F	G
$32 \times 32 \times 32$	218	119	218	288	1	4	266	172	384	493	3	9
$48 \times 48 \times 48$	288	171	355	475	3	11	338	198	399	551	7	31
$96 \times 96 \times 96$	346	210	429	568	18	78	358	211	425	563	53	241
3D FP64 (100 iterations)												
Batch Size	24						72					
	Bandwidth				Energy		Bandwidth				Energy	
Mesh	F	Gx	Gy	Gz	F	G	F	Gx	Gy	Gz	F	G
$32 \times 32 \times 32$	201	165	358	445	2	6	239	193	420	527	6	17
$48 \times 48 \times 48$	242	194	401	536	7	20	267	207	420	554	18	59
$96 \times 96 \times 96$	271	205	426	550	47	155	276	211	442	565	139	464

in relation to (7)). The final term in (15) and (16) are the latencies for processing a batch of B systems. Replacing the reduced system solve with the PCR algorithm is also possible where the $4gt_1$ and $4gt_2$ terms in (15) and (16) then become $log(2t_1) \times (2t_1 + l)$ and $log(2t_2) \times (2t_2 + l)$. Here, l is circuit pipeline latency as discussed in Section 3.

Figure 4 (c) presents 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 `Tridslv(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 for Thomas-PCR; Thomas-Thomas gave similar results). The FPGA can be seen to be 2–3× more energy efficient than the GPU for the largest mesh sizes.

4.2 Stochastic Local Volatility

The second application we evaluate comes from computational finance. It implements a stochastic local volatility (SLV) model, which describe asset price processes, particularly foreign exchange rates [25]. A batched GPU implementation based on a second order finite-difference scheme was developed for this problem using the OPS DSL by Reguly et al. [22]. It is a 2D application implemented in

Table 3: ADI Heat Diffusion App (2D FP32) – Large meshes, Thomas-PCR: 100 iterations, Bandwidth (GB/s), Energy (J) on the FPGA (F) and GPU (G).

Batch Size	60					180				
	Bandwidth			Energy		Bandwidth			Energy	
Mesh	F	Gx	Gy	F	G	F	Gx	Gy	F	G
256^2	206	117	238	5	14	217	186	464	13	35
512^2	218	177	400	17	48	222	210	544	51	128
896^2	220	204	503	53	142	222	214	566	156	418

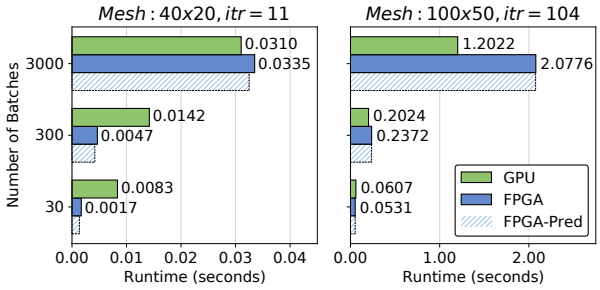


Figure 5: SLV application performance.

FP64 precision. Its high-level algorithm is detailed in Algo. 4. The

Algorithm 4: `2D Heston SLV Backward`

```
for i = 0, i < n_iter, i++ do
    hv_pred0(), hv_matrices()
    Tridslv(x-dim)
    hv_pred1(), Tridslv(y-dim)
    hv_pred2(), Tridslv(x-dim)
    hv_pred3(), Tridslv(y-dim)
end for
```

application implements a Hundsdorfer-Verwer (HV) method, (also based on the ADI method) for time integration. The Rannacher smoothing available in the original application has been switched off in our evaluation. The `hv_pred*` and `hv_matrices` are explicit loops each using 10 point stencils, requiring a window buffer implementation [13] for data reuse. The 9 kernels in Algo. 4 were implemented as separate hardware modules, pipelining the computation within the iterative loop. `hv_matrices` generates a number of 2D coefficients `AX,BX,CX,AV,BV,CV` and 1D coefficient `EV` for the `Tridslv`s. Coefficients `AX,BX,CX` then needs to be input to (consumed by) `Tridslv(x-dim)` kernels and coefficients `AV,BV,CV` and `EV` to `Tridslv(y-dim)` kernels. A GPU implementation consists of these nine kernels, moving data through global memory. Again, it is not possible to fuse kernels to reduce global memory accesses for this bandwidth limited application. The large number of kernels inside the iterative loop incur significant kernel call overhead and data movement through the fixed data path increases latency on the GPU for processing meshes with smaller batch sizes, leading to poor bandwidth utilization.

On FPGA, generated coefficients are consumed at different stages of the pipeline. However other inputs to the `Tridslv` calls come through the computation of this multi-stage pipeline. Therefore

Table 4: SLV App: Bandwidth (GB/s) and Energy (J).

	Bandwidth			Energy	
Batch	FPGA	GPU-x	GPU-y	FPGA	GPU
40×20 mesh: 11 iterations					
30	55.24	3.04	28.01	0.13	0.45
300	202.31	16.48	176.51	0.35	1.02
3000	281.06	123.84	327.65	2.51	4.75
100×50 mesh: 104 iterations					
30	124.63	51.28	109.65	3.98	3.76
300	278.87	235.22	238.34	17.79	22.26
3000	318.36	421.77	429.21	155.82	216.40

large FIFO delay buffers are required to keep synchronization (i.e. avoid pipeline stalling). As such we opt to regenerate the above coefficients at separate stages, essentially duplicating the circuitry. This results in the generation of coefficients `AX,BX,CX`, for the `Tridslv(x-dim)`, being fused to `hv_pred0()` and `hv_pred2()` and the generating of coefficients `AV,BV,CV,EV`, for `Tridslv(y-dim)`, being fused to `hv_pred1()` and `hv_pred3()`. This results in a total of 8 hardware modules, requiring significantly smaller delay buffers than if we implemented the original set of kernels. The performance model for SLV is given in (17):

$$L_{slv} = n_{iter}[4 \times (2x) + 2 \times (3gx) + 2 \times (3gy + 2xy) + \lceil B/N_{CU} \rceil xy] \quad (17)$$

Here g is 64 as SLV uses FP64. The first term is the combined input/output latency for the four explicit stencil computations in `hv_pred*`. The second and third terms account for the `Tridslv (x-dim)` and `Tridslv(y-dim)` calls respectively, including the read or write y-lines from the buffered x-lines. The final term is the latency for processing a batch size of B, 2D meshes. The number of CUs, N_{CU} for SLV on the FPGA is 3, given the considerably larger amount of DSP and memory resources required for the application, particularly due to its use of FP64 precision. The FIFO delay-buffer size 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 [22]. Additionally carrying out extensive speculative scenarios required by regulators under various market conditions to evaluate a bank's exposure means that there are large number 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 detail 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 [22], 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 253MHz. 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 compared to 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 DISCUSSION

The experiments in Section 4.1 show better performance on the Xilinx Alveo U280 FPGA compared to the Nvidia V100 GPU 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 the FPGA although U280 HBM's maximum theoretical bandwidth (460GB/s) is close to half of the V100 HBM (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 8×8 transpose on the FPGA enables higher throughput for `Tridslv(x-dim)` making memory accesses coalesced, 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 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 the SLV application on larger meshes. Future FPGAs with more DSP blocks or floating point primitives will provide better performance than the Xilinx 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.

6 RELATED WORK

Earlier work implementing tridiagonal system solvers on FPGAs such as by Oliveira et al. [19], Warne et al. [29] and Zhang et al. [32] 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. Oliveira et al. [19] pipelined both the forward and backward loops and applied data flow between them and demonstrated the implementation for a smaller 16^3 mesh based application using only on-chip memory.

With the introduction of High-Level synthesis (HLS) tools, a number of more recent works [16–18, 30] implemented the Thomas, PCR, and Spike algorithms on FPGA using HLS tools. Many of these did not demonstrate the solver working on full applications, with the exception of László et al. in 2015 [16] which compared a one factor Black-Scholes option pricing equation using explicit and implicit methods on different architectures such as multi core CPUs, GPUs, and FPGAs. Their implementation, based on the Thomas algorithm, targets a Xilinx Virtex 7 FPGA and effectively pipelines

both forward and backward loops but was not able to apply data flow between these two steps and results showed an Nvidia K40 GPU significantly outperforming the FPGA.

Macintosh, et al. [18] used OpenCL targeting an Altera Stratix V FPGA to implement the PCR and SPIKE algorithms, showing comparable performance to an Nvidia Quadro 4000 GPU, not including reconfiguration time for the spike kernels. Later, Macintosh, et al. [17] used OpenCL to develop `oclspkt`, a library that implements tridiagonal systems solvers targeting FPGAs, GPUs, and CPUs. `oclspkt` uses the truncated spike algorithm for diagonally dominant tridiagonal matrices, and as such does not give exact solutions. Their results show `oclspkt` on an Altera Arria 10GX FPGA performing marginally slower than an Nvidia Quadro M4000 GPU but providing better energy efficiency. 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 [13], high throughput for small and medium sized systems can be achieved. The Thomas algorithm uses fewer 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 Thomas-PCR solvers to handle a number of partitioned systems and then a reduced system solve to exploit the limited available on-chip memory resources of a single FPGA.

Several recent works have also exploited HBM in modern FPGAs [12, 14, 24] showing performance gains and energy savings for memory bandwidth bound applications compared to traditional architectures and FPGA devices without HBM. Multi-dimensional tridiagonal solvers are also bandwidth bound, but, to our knowledge, no previous work has explored the use of HBM capable FPGAs to accelerate them, as we have done through the use of parallel compute units. To our knowledge, the 2D/3D ADI and SLV applications developed in this work, motivated by real-world implicit problems on FPGAs are also novel; SLV being one of the few non-trivial applications using multi-dimensional tridiagonal solvers presented in the literature. The Thomas based solver developed in this paper gives higher performance than the current PCR based Xilinx library, as shown in Section 4. Additionally, the analytical performance model and the 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.

7 CONCLUSION

We have 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 better 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 considerable performance 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 an analytical performance model was developed to estimate runtime performance of the FPGA designs and 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 operational cost benefit. The analytical performance 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 performance on FPGAs, as demonstrated in this work, provide key insights into the feasibility and profitability of using FPGAs in high performance computing workloads.

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 further support the conclusions in this paper. 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

[1] 2020. Nvidia V100 Data Sheet. https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf.
[2] 2020. Tridsolver Library. https://github.com/OP-DSL/tridsolver.
[3] 2020. Vitis High-Level Synthesis User Guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf.

[4] 2020. Vitis Quantitative Finance Library V.2020.2. https://xilinx.github.io/Vitis_Libraries/quantitative_finance/2020.2/.

[5] 2021. Batched Tridiagonal Systems Solver Library for FPGAs. [https://github.com/Kamalavasan/Tridsolver-FPGA].

[6] 2021. cuSPARSE API Reference. [https://docs.nvidia.com/cuda/cusparse/index.html].

[7] D. Balogh, T. Flynn, S. Laizet, G. R. Mudalige, and I. Reguly. 2021. Scalable Many-Core Algorithms for Tridiagonal Solvers. *Computing in Science and Engineering* 24, 01 (2021), 26–35.

[8] Johannes de Fine Licht, Andreas Kuster, Tiziano De Matteis, Tal Ben-Nun, Dominic Hofer, and Torsten Hoefler. 2021. StencilFlow: Mapping Large Stencil Programs to Distributed Spatial Computing Systems. , 315–326 pages.

[9] Tiziano De Matteis, Johannes de Fine Licht, and Torsten Hoefler. 2020. FBLAS: Streaming Linear Algebra on FPGA. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 59, 13 pages.

[10] Jim Douglas and James E Gunn. 1964. A General Formulation of Alternating Direction Methods. *Numèrische mathèmatik* 6, 1 (1964), 428–453.

[11] Walter Gander and Gene H Golub. 1997. Cyclic Reduction—History and Applications. *Scientific computing (Hong Kong)* 7385 (1997).

[12] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. 2021. GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*.

[13] Kamalavasan Kamalakkannan, Gihan R. Mudalige, István Z. Reguly, and Suhaib A. Fahmy. 2021. High-Level FPGA Accelerator Design for Structured-Mesh-Based Explicit Numerical Solvers. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1087–1096.

[14] Kaan Kara, Christoph Hagleitner, Dionysios Diamantopoulos, Dimitris Syrivelis, and Gustavo Alonso. 2020. High Bandwidth Memory on FPGAs: A Data Analytics Perspective. In *International Conference on Field-Programmable Logic and Applications (FPL)*.

[15] Endre Laszlo, Mike Giles, and Jeremy Appleyard. 2016. Manycore Algorithms for Batch Scalar and Block Tridiagonal Solvers. *ACM Transactions on Mathematical Software (TOMS)* 42, 4 (2016), 1–36.

[16] Endre László, Zoltán Nagy, Michael B. Giles, István Reguly, Jeremy Appleyard, and Peter Szolgay. 2015. Analysis of Parallel Processor Architectures for the Solution of the Black-Scholes PDE. In *IEEE International Symposium on Circuits and Systems (ISCAS)*. 1977–1980.

[17] H. Macintosh, Jasmine Banks, and N. Kelson. 2019. Implementing and Evaluating an Heterogeneous, Scalable, Tridiagonal Linear System Solver with OpenCL to Target FPGAs, GPUs, and CPUs. *International Journal of Reconfigurable Computing* (2019).

[18] H. J. Macintosh, D. J. Warne, N. A. Kelson, J. E. Banks, and T. W. Farrell. 2016. Implementation of Parallel Tridiagonal Solvers for a Heterogeneous Computing Environment. In *Proceedings of the Biennial Computational Techniques and Applications Conference (CTAC)*, Vol. 56. C446–C462.

[19] Filipe Oliveira, C. Silva Santos, F. A. Castro, and José C. Alves. 2008. A Custom Processor for a TDMA Solver in a CFD Application. In *Reconfigurable Computing: Architectures, Tools and Applications*, Roger Woods, Katherine Compton, Christos Bouganis, and Pedro C. Diniz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 63–74.

[20] Donald W Peaceman and Henry H Rachford, Jr. 1955. The Numerical Solution of Parabolic and Elliptic Differential Equations. *Journal of the Society for industrial and Applied Mathematics* 3, 1 (1955), 28–41.

[21] Eric Polizzi and Ahmed H. Sameh. 2006. A Parallel Hybrid Banded System Solver: the SPIKE Algorithm. *Parallel Comput.* 32, 2 (2006), 177–194.

[22] Istvan Z. Reguly, Branden Moore, Tim Schmielau, Jacques du Toit, and Gihan R. Mudalige. 2019. Batch solution of small PDEs with the OPS DSL. In *High Performance Computing*, Michèle Weiland, Guido Juckeland, Sadaf Alam, and Heike Jagode (Eds.). Springer International Publishing, Cham, 124–141.

[23] Bajaj Ronak and Suhaib A Fahmy. 2015. Mapping for maximum performance on FPGA DSP blocks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 4 (2015), 573–585.

[24] Gagandeep Singh, Dionysios Diamantopoulos, Christoph Hagleitner, Juan Gomez-Luna, Sander Stuijk, Onur Mutlu, and Henk Corporaal. 2020. NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling. In *International Conference on Field-Programmable Logic and Applications (FPL)*. 9–7.

[25] G. Tataru and T. Fisher. 2010. Stochastic Local Volatility. *Quantitative Development Group, Bloomberg Version 1* (Feb 2010).

[26] Llewellyn Thomas. 1949. Elliptic Problems in Linear Differential Equations Over a Network: Watson Scientific Computing Laboratory. *Columbia Univ., NY* (1949).

[27] Pedro Valero-Lara, Ivan Martínez-Pérez, Raül Sirvent, Xavier Martorell, and Antonio J. Peña. 2018. Nvidia GPUs Scalability to Solve Multiple (Batch) Tridiagonal Systems Implementation of cuThomasBatch. In *Parallel Processing and Applied Mathematics*, Roman Wyrzykowski, Jack Dongarra, Ewa Deelman, and Konrad Karczewski (Eds.). Springer International Publishing, Cham, 243–253.

[28] Xinliang Wang, Yangtong Xu, and Wei Xue. 2014. A Hierarchical Tridiagonal System Solver for Heterogeneous Supercomputers. In *Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. 69–76.

[29] David Warne, Neil Kelson, and Ross Hayward. 2012. Solving Tri-diagonal Linear Systems Using Field Programmable Gate Arrays. In *Proceedings of the International Conference on Computational Methods*, Y Gu and S Saha (Eds.). Queensland University of Technology, Australia.

[30] David J. Warne, Neil A. Kelson, and Ross F. Hayward. 2014. Comparison of High Level FPGA Hardware Design for Solving Tri-diagonal Linear Systems. *Procedia Computer Science* 29 (2014), 95–101.

[31] Xilinx Inc. 2020. *Alveo U280 Data Center Accelerator Card Data Sheet.* Xilinx Inc. v1.3.

[32] Wei Zhang, Vaughn Betz, and Jonathan Rose. 2012. Portable and Scalable FPGA-Based Acceleration of a Direct Linear System Solver. *ACM Trans. Reconfigurable Technol. Syst.* 5, 1, Article 6 (March 2012), 26 pages.