FPGA Acceleration of Structured-Mesh-Based Explicit and Implicit Numerical Solvers using SYCL

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

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

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

ABSTRACT

We explore the design and development of structured-mesh based solvers on current Intel FPGA hardware using the SYCL programming model. Two classes of applications are targeted : (1) stencil applications based on explicit numerical methods and (2) multi-dimensional tridiagonal solvers based on implicit methods. Both classes of solvers appear as core modules in a wide-range of real-world applications ranging from CFD to financial computing. A general, unified workflow is formulated for synthesizing them on Intel FPGAs together with predictive analytic models to explore the design space to obtain near-optimal performance. Performance of synthesized designs, using the above techniques, for two non-trivial applications on an Intel PAC D5005 FPGA card is benchmarked. Results are compared to performance of optimized parallel implementations of the same applications on a Nvidia V100 GPU. Observed runtime results indicate the FPGA providing better or matching performance to the V100 GPU. However, more importantly the FPGA solutions provide 59%-76% less energy consumption for their largest configurations, making them highly attractive for solving workloads based on these applications in production settings. The performance model predicts the runtime of designs with high accuracy with less than 5% error for all cases tested, demonstrating their significant utility for design space explorations. With these tools and techniques, we discuss determinants for a given structured-mesh code to be amenable to FPGA implementation, providing insights into the feasibility and profitability of a design, how they can be codified using SYCL and the resulting performance.

KEYWORDS

FPGA, SYCL, Stencil Applications, Explicit solvers, Implicit Solvers

ACM Reference Format:
K. Kamalakkannan, G.R. Mudalige, I.Z. Reguly, and S.A. Fahmy. 2022. FPGA Acceleration of Structured-Mesh-Based Explicit and Implicit Numerical Solvers using SYCL. In *IWOCL and SYCLcon, May 12–12, 2022, Virtual.* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Field Programmable Gate Arrays (FPGAs) have gained traction as accelerator devices, particularly due to their low power consumption and re-programmability. Their utility in accelerating scientific computing workloads have recently been gaining interest in the high performance computing (HPC) community with several studies showing competitive or better performance compared to traditional CPU/GPU architectures, for select classes of applications [9, 13, 19, 27].

Compared to CPUs or GPUs, FPGAs do not have a fixed general purpose architecture to be programmable with software. Instead, a tailored data-path for the computation is synthesized using a variety of basic circuit elements. However, to gain good performance, significant effort is needed, for example by creating pipelined custom datapaths, replication of compute units and exploiting locality such that data can be reused by fitting on to fast on-chip memory. FPGA hardware designers/vendors such as Xilinx and Intel have attempted to support the development effort with high-level synthesis tools (HLS) that can translate high-level languages such as C/C++ or OpenCL, producing designs targeting FPGAs. However, these tools still require low-level modifications and specialized handling of the codes to produce performant designs. A recent addition to the high-level programming models for FPGAs is Intel's OneAPI, itself based on the SYCL programming model. Currently, it mainly supports (at a production level) the programming of Intel FPGA hardware and aims to provide a unifying model for programming Intel's many hardware offerings, from Xeon CPUs, Xe GPUs and FPGAs. Given the novelty of SYCL and its use for programming FPGAs, the underlying goal of this paper is to investigate how SYCL can be used for obtaining the best performance from Intel's FPGA devices. We make the following specific contributions:

(1) Based on design strategies developed in previous work [19, 20], in this paper we develop a generalized workflow for formulating optimized FPGA designs with SYCL, targeting Intel FPGAs for two classes of structured-mesh based solvers: (1) stencil applications based on explicit numerical methods and (2) multi-dimensional tridiagonal solvers based on implicit methods. Given FPGA resource constraints, we exploit features of the application classes, that lend to gaining higher optimizations. These include the iterative nature of the numerical methods, particularly in implementing explicit/implicit solvers and as observed in many production settings, the need to solve large numbers of independent problems on multiple meshes, leading to batched solves. Predictive analytic models are developed alongside the above workflows to support the design space explorations.

(2) Two non-trivial applications, a 3D Reverse Time Migration (RTM) Forward Pass solver based on explicit methods and a 2D Alternating Direction Implicit (ADI) problem solving the heat diffusion equation using multi-dimensional tridiagonal solvers, are designed and synthesized on an Intel PAC D5005 using the above techniques. Their runtime, bandwidth and power/energy performance is benchmarked and compared to highly optimized implementations of the same applications on a GPU (a Nvidia V100 GPU), the currently best performing processor architecture for structured-mesh applications. We demonstrate the use of SYCL in achieving the desired circuit synthesis, with challenges and opportunities in the programming model to elucidate a design on hardware.

Initial results on the Intel PAC D5005 demonstrate the FPGA providing performance better or on-par to that of the V100 GPU in terms of runtime. More importantly the FPGA solutions provide 59%-76% less energy consumption for their largest configurations, making them highly attractive for solving workloads based on these applications in production settings. The performance models provided high accuracy with less than 5% model prediction errors for all cases, demonstrating their significant utility for design space explorations. The techniques developed previously in [19, 20] were explored in the context of Xilinx FPGAs, the other currently dominant FPGA device in the market. With the present work, we broaden these generalized designs to include features of Intel FPGAs enabling to provide a comprehensive workflow for design-space explorations. We believe that taken together, this simplifies and standardizes the development cycles for industrial workloads consisting of these application classes, particularly from areas such as the financial computing domain for FPGA hardware.

The rest of this paper is organized as follows: Section 2 begins with background on structured mesh solvers, both explicit and implicit, their underlying algorithms together with previous work that explored FPGA implementations for this class of applications. Section 3 and Section 4 presents the main contributions, detailing the design extensions for each class of applications. Section 5 details the experimental results, specifically performance results. Finally, Section 6 presents conclusions from the work.

2 BACKGROUND

2.1 Explicit and Implicit Methods

Computations on a structured mesh can be most commonly characterized as calculations performed on a "rectangular" multi-dimensional set of mesh points. The regular nature of the domain, therefore allows the neighborhood of a mesh point to be easily computed, the most common form being the fixed data access pattern provided by a *stencil*. This is in contrast to unstructured-mesh applications [8] where explicit mesh connectivity information is required to implement computations over mesh elements. The main motivating numerical method for stencil computations is the solution to Partial Differential Equations (PDEs), specifically based on the finite difference method. These techniques are used extensively in computational fluid dynamics (CFD), computational electromagnetics (CEM) in the form of iterative solvers. For example the finite difference scheme for the solution of a generic PDE can be given by the 2D explicit equation (1):

$$U^{t+1}_{x,y} = aU^t_{x-1,y} + bU^t_{x+1,y} + cU^t_{x,y-1} + dU^t_{x,y+1} + eU^t_{x,y} \quad (1)$$

Here, U is a 2D mesh and *a, b, c, d,* and *e* are coefficients. In this example U is accessed at spatial mesh points *(x-1,y), (x+1,y), (x,y-1), (x,y+1),* and *(x,y)* which forms a five point stencil. The above scheme is noted as an explicit scheme where the computation iterates over the full rectangular mesh, updating the solution at each mesh point, for the current time step, *t+1*, using the solution from the previous time step, *t*. The time step iteration usually continues until a steady state solution is achieved. In contrast an implicit scheme would update the solution at the current time step using values from the same time step. Thus, in explicit methods, there is a data dependency only for the computations among multiple time step iterations where each mesh point calculation within a time iteration can be computed in parallel. However, in implicit methods a further dependency in the spatial domain is also introduced. This may lead to a much faster convergence to the final solution, but enforces an order in which a computation iterates over the mesh leading to limited parallelism.

Considering implicit schemes, one used extensively in areas such as financial computing is the solution to multi-dimensional tridiagonal systems, which we focus on in this paper. Tridiagonal solvers are frequently used in the Alternating Direction Implicit (ADI) time discretization problems [14, 28] preferred in industry. Tridiagonal systems arise from the need to solve a system of linear equations as given in equation (2), where $a_0 = c_{N-1} = 0$. Its matrix form $Ax = d$ can be stated as in equation (3).

$$a_i u_{i-1} + b_i u_i + c_i u_{i+1} = d, \quad i = 0, 1, ..., N-1 \quad (2)$$

$$\begin{bmatrix} b_0 & c_0 & 0 & \dots & 0 \\ a_1 & b_1 & c_1 & \dots & 0 \\ 0 & a_2 & b_2 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{N-1} & b_{N-1} \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{N-1} \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{N-1} \end{bmatrix} \quad (3)$$

The solution to such systems of equations are well known, including the Thomas algorithm [36] which carries out a specialized form of Gaussian elimination, the PCR algorithm [16] and the Spike algorithm [29].

Many previous works for synthesizing stencil computations on FPGAs, have been presented. Early works such as [33–35] used low-level Hardware Description Languages (HDL) for describing the architectures. Modern HLS tools based work such as [10, 11, 13, 19, 21, 30, 37, 38, 40] have showcased key workflows and optimizations in synthesizing stencil applications using high-level languages such as C++ and OpenCL on a variety of FPGAs from Xilinx to Intel. Similarly, implicit schemes based solvers such as tridiagonal systems solvers have been implemented in works such as [20, 23–25, 39] including for multi-dimensional multiple tridiagonal solvers in addition to vendor supplied libraries such as [3]. In our present work, the aim is to investigate how the SYCL programming model can be used to synthesise high-performance implementations of these solvers on Intel FPGAs and showcase the challenges and opportunities to elucidate solutions using this new programming model.

```
using namespace sycl;
void stencil_WI( queue &q,
              buffer<float,2> b_data_in,
              buffer<float,2> b_data_out,
              int size0, int size1,
              int block0, int block1){
  q.submit([&] (handler& h){
    accessor in(b_data_in, h);
    accessor out(b_data_out, h);

    range<2> local_range(block0, block1);
    range<2> global_range(size0, size1);

    h.parallel_for<class stencil_WI>
    (nd_range<2>(local_range, global_range),
    [=] (nd_item<2> point){
      int x = point.get_global_id(0);
      int y = point.get_global_id(1);
      if(x > 0 && y > 0 && x < size0-1 && y < size1-1){
        float r = (in[x][y-1] + in[x][y+1])*0.125f +
          in[x][y]*0.5f;
        out[x][y] = r;
      }
  });});
}
```

Figure 1: NDRange based stencil computation

2.2 Intel FPGAs and SYCL

An FPGA device in general, consists of basic circuit elements such as configurable logic known as adaptive logic modules (ALMs) that include lookup tables (LUT) and registers, specialized blocks such as random-access-memory blocks (640-bit MLABs and 20K bits M20K) and Digital Signal Processing (DSP) blocks. These are interconnected via a rich routing fabric providing large bandwidth between elements. A computation/algorithm is implemented on an FPGA by combining these circuit elements, designing a custom data-path through them, essentially setting up a hardware pipeline. In addition to on-chip memory, FPGAs come with large external memory and in modern devices and a further block of memory consisting of 3D stacked High Bandwidth Memory (HBM2). Managing the movement of data between these different types of memory is key to achieving high performance. Basic mathematical operations are generally implemented using the DSP units and a single unit can do one 27×27 multiplication or two 18×18 multiplication per clock. Double precision (FP64) accumulation is also supported. In addition, Intel FPGAs include DSP blocks which also support low latency single precision (FP32) `ADD`,`SUB`,`MUL` and `ACCU` operations. As a given circuit design grows and begins to occupy a larger portion of the FPGA, routing (i.e. connecting all the circuit elements together) becomes more challenging, and can reduce the achievable clock frequency and hence overall performance. In this aspect, Intel's Hyperflex technology [17] gives more flexibility in routing which helps achieve better clock frequency.

The recently introduced Data Parallel C++ (DPC++) programming model[1] based on the SYCL programming model to program FPGAs follows OpenCL. Here, the portions of the program to be

[1]We use the terms DPC++ and SYCL interchangeably in this work due to the use of Intel target hardware.

```
using namespace sycl;
void stencil_ST(queue &q,
                buffer<float,2> &b_data_in,
                buffer<float,2> &b_data_out,
                int size0, int size1){
  q.submit([&] (handler& h){
    accessor  in(b_data_in, h);
    accessor  out(b_data_out, h);
    h.single_task<class stencil_ST> ([=] (){
      /* optimisations - see Section 2 ... */
      float window1[1024];
      float window2[1024];
      [[intel::loop_coalesce(2)]]
      /* +1 due to one row delay through window buffer */
      for(int y = 0; y < size1+1; y++){
        for(int x = 0; x < size0; x++){
          float s_12;
          if(y < size1) s_12 = in[x][y];
          float s_11 = window1[x];
          float s_10 = window2[x];
          window1[x] = s_12;
          window2[x] = s_11;
          float r = (s_10 + s_12)*0.125f + s_11*0.5f;
          if(x > 0 && y > 0 &&
             x < size0-1 && y < size1){
             out[x][y-1] = r;
          }
        }
      }
  });});
}
```

Figure 2: single_task based stencil computation

executed on an accelerator device is called a *kernel*. SYCL's accelerator model consists of a number of compute units, each made of processing elements (e.g. SMs on a GPU). An instance of a kernel executed on a processing element is called a *work-item* (equivalent to *threads* in the CUDA programming model) and an instance of a work-item is identified using an index *id* in a global index space. Work-items are organized into groups called *work-groups* (*thread-blocks* in CUDA) and each work-item inside the group will have a local-id. Work-items in a work-group will be executed concurrently on processing elements of the compute unit where the index space is specified using SYCL's N-dimensional range model. Kernels based on this index space are called `NDRange` kernels (see Figure 1).

`NDRange` kernels therefore essentially follow a single instruction multiple thread (SIMT) execution model where on a GPU, multiple kernels are called, with the system scheduling them to be executed on the available SMs. Such an execution can be done for FPGAs as well, but performance becomes severely limited due to FPGA global memory bandwidth (about 19 − 76GB/s with DDR4 or ≈ 460GB/s with HBM2, compared to over 900GB/s on modern GPUs), when having to write results of one kernel back to global memory before calling the next kernel and the new kernel having to read all the necessary data back from global memory. However, on-chip memory bandwidth on FPGAs reaches over tens of TB/s and therefore feeding the results from one kernel to the next in a pipeline, provides significant opportunities for performance gains. This is achieved by a *single task* kernel, by attempting to create longer and longer computational pipelines essentially following a multiple instructions single data (MISD) model. A sequence of kernels within nested loops then leads to flattening the loop nests and fusing the loops (see discussion in Section 3), of course within the resource limits of the FPGA for implementing the computation. With SYCL, such a single task kernel can be codified as in Figure-2.

In comparison to OpenCL, which is used by most of the previous work on Intel FPGAs [18, 24, 25, 37, 38, 40, 41], SYCL provides a higher level of abstraction, including removing much of the fixed "boiler-plate" code segments required to setup the device. Additionally, kernel arguments need not be set explicitly and data is automatically moved from host to device through `sycl::buffers`. Device memory is released when these buffers run out of scope. Essentially, the SYCL run-time makes sure that data is available on device/host before the execution of the kernel/host part of program. The runtime additionally analyzes data dependencies where a kernel consuming dependent data will not be scheduled for execution until completion of kernels that produce that data. This introduces the limit of only one kernel being able to write to a data structure at a time. An example of issues due to this limitation include the challenge of moving the time-marching loop in an explicit stencil computation to the FPGA. This and other key designs for synthesizing the two classes of applications on Intel FPGAs using SYCL are discussed in the next two sections.

3 STENCIL SOLVERS

Previous work in [19] developed a generalized workflow for synthesizing stencil applications on Xilinx FPGAs. In this section we build on this design flow to extend it to Intel FPGAs with SYCL. The full FPGA designs implemented with SYCL can be found in [4].

Nested Loop Unrolling: The workflow in [19], considered a given stencil application as specified by equation (1), to be a sequence of repeating nested loops iterating over the mesh. The repeating of nested loops, as discussed before, occurs due to the implementation of a time-marching outer loop. On an FPGA, a single nested loop's body, consisting of the elemental computation per iteration will need to be synthesized using the basic circuit elements we noted before. This will essentially create a circuit pipeline, through which input data will flow through, with each clock cycle of the device. The pipeline will need to be started up, requiring some clock cycles equal to the depth of the pipeline (which depends on the complexity of the elemental computation synthesized) and outputting the result from the computation for each inner iteration. For best performance we would like to get an output per clock cycle. However, encompassing this elemental computation in multiple levels of the loop nest can be detrimental to performance due to the need to flush the results from the inner loop, which can take a long time, leading to a stall in the pipeline when moving to successive outer iterations of the loop nest[11]. As we alluded to previously, a multi-dimensional nested loops should therefore be flattened to a 1D loop either manually or by using HLS directives. With SYCL (specifically with the `Intel`® `oneAPI DPC++/C++` compiler, Intel's implementation of a SYCL compiler), we can easily achieve this by using the `loop_coalesce` attribute specified as a pragma to the nested loop.

Vectorisation (Cell Parallel method): Replicating the circuitry for the elemental computation can also be done, provided (1) there

```
/* Data type for wider data path */
struct dPath16 {[[intel::fpga_register]]float data[16];};

for(int i = 0; i < total_itr; i++){
 struct dPath16 s_1_0, s_1_1, s_1_2, vec_wr;
 /* other declarations, index calculation, window buffer*/
 #pragma unroll VFACTOR
 for(int v = 0; v < VFACTOR; v++){
  int i_ind = i *VFACTOR + v;
  float val = (s_1_0.data[v]+s_1_2.data[v])*0.125f+ \
           s_1_1.data[v]*0.5f;
  vec_wr.data[v] = (i_ind >0 && i_ind<nx-1 && j>1 && j<ny ) \
                 ? val : s_1_1.data[v];
  }
  /* writing results to pipe */
}
}
```

Figure 3: vectored stencil computation

are no data dependencies between the nested loop iterations and (2) there are enough resources available for synthesis on the FPGA (e.g. DSP units, FP cores etc). For stencil applications there are no such dependencies. This will lead to multiple pipelines (number limited by resources) operating in parallel, similar in operation to a vector operation on CPUs. This technique is also known as the *cell-parallel* method [37, 38], where if you visualize the stencil computation implemented with a nested loop as a loop over a regular multi-dimensional rectangle of mesh points/cells, this method will compute multiple "cells" in parallel. For SYCL synthesis, this requires reading a wider block of memory, we prefer to program with a struct based data type as in Figure 3.

Window Buffers: Next, the design flow from [19] specifies the need to stream data from/to external (DDR4) and near-chip (HBM2) memories to/from on-chip MLABs/M20Ks to feed the computational pipeline efficiently. A perfect data reuse path can be created by (1) using a First-In-First-Out (FIFO) buffer to fetch data from DDR4/HBM memory without interruption (allowing burst transfers) to on-chip memory, and then (2) by caching mesh points using the multiple levels of memory, from registers to MLABs/M20Ks. This is known as implementing a *window buffer* [15]. To implement such a data-reuse path with SYCL, the external/near-chip memory read, computation and write back to external/near-chip memory each were codified as a separate SYCL kernel with `sycl::pipes`, a DPC++ extension, used to move data between the kernels. These kernels operate in parallel. A basic window buffer setup can be seen in Figure 2 lines 11-12 and lines 17-20 where we are reading and writing values such that a certain length of data is buffered in the window.

Unrolling Iterative (time-marching) loop / Step Parallel method: Next step in the design strategy is to unroll the iterative (time-marching) loop which can include one or more stencil loops over the rectangular mesh. This allows the results from a previous iteration (time-step) to be input into the next iteration without needing to write results back to external memory. This technique has been called the *step-parallel* method where the unrolling yields multiple compute modules, again replicating circuitry. This technique leads to increased throughput without the need for additional external memory bandwidth, but the loop unrolling factor depends

```
1  using namespace sycl;
2  template <int id> struct stencil_compute_id;
3  template<int idx, int DMAX, int VFACTOR>
4  void stencil_compute(queue &q, int size0, int size1){
5   q.submit([&] (handler& h){
6   h.single_task<class stencil_compute_id<idx>> ([=] (){
7    ... // declarations and setups
8    for(int i = 0; i < size0*(size1+1); i++){
9     if(cond1) vec_r =pipeS::PipeAt<idx>::read();
10    ... // window buffers and stencil computation
11    if(cond2) pipeS::PipeAt<idx+1>::write(vec_w);
12   }
13  });});
14 }
```

Figure 4: stencil compute kernel skeleton

```
template <int N> struct itr_loop {
  static void instantiate(queue &q, int nx, int ny){
    itr_loop<N-1>::instantiate(q, nx, ny);
    stencil_compute<N-1, 4096, 8>(q, nx, ny);
  }
};
template<> struct itr_loop<1>{
  static void instantiate(queue &q, int nx, int ny){
    stencil_compute<0, 4096, 8>(q, nx, ny);
  }
};
```

Figure 5: Pipelining stencil compute kernels

once more on available FPGA resources and also internal memory capacity [19].

Unrolling the iterative loop will instantiate the same stencil loop's kernel many times. Thus we use a template based stencil computation function to produce unique names (see Figure 4, line 2). While unique kernel names are optional in SYCL 2020, unique kernel names are used here to create multiple instance of same kernels on FPGA. As noted before, to move data from one kernel to another, internally, SYCL uses pipes [7] which are similar to streams (e.g. `hls::stream`) in Vivado C++ on Xilinx FPGAs. Thus, a stencil kernel will get input from a pipe and it will push the output to another pipe. As such, pipes should also be unique to indicate the connection between the unique producer/consumer kernels. An indexable pipe array can be created using a struct construct to obtain unique pipes. We use the index from the instantiated template of the stencil compute function for kernels name and choose the pipes as illustrated in Figure 4.

Unrolling of the time-marching loop can be implemented in SYCL using a template based recursive struct function and with a template specialization as in Figure 5. Here we note that, `stencil_compute` kernel pops the input data from and pushes the result to the relevant pipes with the adjacent index on the pipe array. Using these techniques we can create a kernel pipeline with any given iterative loop unroll factor of N.

Batching: The above compute kernel pipeline with read/write kernels and unrolled iterative loop will give good performance for larger mesh sizes which hides the kernel call time on the device. However, for smaller meshes which usually have smaller execution times the kernel calling overhead is a significant portion of the

```
[[intel::disable_loop_pipelining]]
for(int itr = 0; itr < n_iter; itr++) {
  accessor ptrR = ((itr & 1) == 0) ? in : out;
  accessor ptrW = ((itr & 1) == 1) ? in : out;
  [[intel::ivdep]] [[intel::initiation_interval(1)]]
  for(int i = 0; i < total_itr+delay; i++) {
    struct dPath16 vecR = ptrR[i+delay];
    if(i < total_itr) pipeM::PipeAt<idx1>::write(vecR);
    struct dPath16 vecW;
    if(i >= delay) vecW = pipeM::PipeAt<idx2>::read();
    ptrW[i] = vecW;
  }
}
```

Figure 6: Global memory read-write loop

total runtime. Additionally, on FPGAs a further overhead is caused by the latency of the processing pipeline compared to the time to process the mesh. A key development from [19] is to amortize these overheads by executing a larger number (a batch) of smaller meshes. Batching will then involve grouping together meshes with the same dimensions, leading to increased overall throughput of problems solved per time unit. As indicated in [19], in practice, the mesh can be extended in the last dimension by stacking up the small meshes. Now, compute latencies for pipeline startup also only occur once at the start of the batched solve. No special techniques in SYCL is required to program batching and we discuss and quantify the performance implications of batching later in section 3.1

Reducing kernel call overhead: While batching provides a reasonably good way to hide kernel call overheads, it really needs large batch sizes to be effective. For example for the RTM application, we benchmark later in the paper, a 3D mesh of size $32 \times 32 \times 32$ require a batch size of 1000 (i.e. 1000 meshes) to hide kernel call latency. A general solution for this problem is to move the time-marching loop to the FPGA [19]. When the host executes the time-marching loop, the read and write kernels are called by swapping the memory locations and the runtime can schedule the read kernel and write kernel at the same time as each access different data structures. In this case the host has to wait until completion of the dependant kernels, providing an implicit sync point. However, if the time-marching loop is moved to the device (i.e. FPGA), then both the read and write kernels will need to be called with both the read and write memory locations together with a signal/flag (through a pipe) to notify the read kernel that the write kernel has completed writing to the specified memory locations. In this case the SYCL runtime will note this as a data dependency, leading to a deadlock. The run-time will wait for write module to complete first before scheduling the read kernel, hanging the kernel pipeline. In contrast on Xilinx FPGAs using C++ for Vivado [19], such a deadlock will not occur as a more hand-tuned complete data-flow path can be created from read, compute to write within the iterative loop in a single kernel.

A solution can be attempted to avoid a deadlock, by fusing global memory read and write accesses into one nested loop as in Figure 6, creating one single kernel. Attribute `intel::ivdep` is used to instruct the compiler that there are no memory access dependencies in the inner loop allowing the compiler to fully pipeline the inner loop. Pipelining is disabled for outer loop due to data dependency between iterations as the inner loop's read and write locations are swapped in each iteration. However, this implementation will also result in a deadlock or poor performance if pipe `read` is not delayed correctly. To understand the issues we need to look at how statements inside a kernel are scheduled and how an iteration of a loop moves through the stages of the compute pipeline for each clock cycle on the FPGA.

In Figure 6, Pipe `read` and `write` are blocking operations, where the loop iteration will not continue until these operations complete. Here `vecR` is loaded from memory and will be pushed to the pipe and will go through the compute pipeline before returning as an output result `vecW` through another pipe for `read`. There are number of clock cycles between the first push of `vecR` and first pop of `vecW`. Assume for example `read` (*rd*) and `write` (*wr*) are scheduled at the 10^{th} and 800^{th} clocks (exact clocks at which *rd* and *wr* are scheduled could be obtained from the kernel schedule viewer section of the report generated by Intel's DPC++ compiler). Then for loop iteration 0, `rd` and *wr* operations are scheduled at 10^{th} and 800^{th} clocks and for iteration 1 they are scheduled at 11^{th} and 801^{th} clocks and so on. If first *rd* is not successful until the 50^{th} clock cycle, then first `wr` can only occur at the 840^{th} clock. Such a blocking can be avoided by introducing a delay as done in the conditional statement at line 10 in Figure 6.

To calculate the required delay, assume *rd, wr* operations are scheduled at clk_{rd} and clk_{wr} and there are S number of pipeline stages between pipe `idx1` and pipe `idx2`. In this case, data pushed to pipe `idx1` will come back to pipe `idx2` only after S clock cycles. Hence, we have to delay the pipe `read` by delay $d >= clk_{rd} - clk_{wr} + S$ number of loop iterations to avoid stalling of pipeline (here we assume uniform data path width across kernels). If delay $d >= clk_{rd} - clk_{wr}$ and $d < clk_{rd} - clk_{wr} + S$ then there will be stalling, but loop will continue as data will be available after a fewer clock cycles than expected, leading to reduced throughput. In case of delay $d < clk_{rd} - clkwr$, then the implementation will deadlock as data is expected from `read` pipe (`idx2`) before it is pushed to the pipe `idx1`. A further consideration for a 2D stencil computation as in Figure 2, is that at least a row of elements are required to start the computation and return the first output. This adds an additional delay which we note as a buffer delay d_b, leading to a total delay $d >= clk_{wr} - clk_{rd} + S + d_b$ to avoid stalling. Again $d < clk_{wr} - clk_{rd} + d_b$ will result in a deadlock as pushed data will never be available at the time a `read` is attempted and it stalls the whole loop iteration leading to no new data also being pushed to the `write` pipe.

3.1 Performance Model

The full design for 2D and 3D stencil applications can be modeled analytically to predict the total runtime of the time-marching loop. In [19], the model for implementations on Xilinx FPGAs were developed and the same terms can be used here for our SYCL design on Intel FPGAs with the addition of the schedule delays discussed in the previous section. Schedule delays are usually small when the mesh size is reasonably large but it is significant compared to the processing time for smaller meshes with small batch sizes. The total delay for a 2D stencil application can be modeled as follows:

$$delay_{2D} = \left(S_{2D} + d_{b,2D}\right) \tag{4}$$

$$S_{2D} = \sum_{i=0}^{kernels} \left(clk_{wr,i} - clk_{rd,i}\right) \tag{5}$$

$$d_{b,2D} = \left\lceil \frac{m}{V} \right\rceil \times p \times \frac{D}{2} \tag{6}$$

Here the mesh size in each dimension is given by m, n and V, p, D are vectorization factor, iterative loop unroll factor and stencil order respectively. $clk_{wr,i}, clk_{rd,i}$ are the clock cycles where pipe `write` and pipe `read` are scheduled in the i^{th} kernel. Here we note that, pipe width is V for all kernels. Similarly for a 3D application the delay can be modeled as in equation (7) when the size of the 3rd dimension is l.

$$delay_{3D} = \left(S_{3D} + d_{b,3D}\right) \tag{7}$$

$$S_{3D} = \sum_{i=0}^{kernels} \left(clk_{wr,i} - clk_{rd,i}\right) \tag{8}$$

$$d_{b\_3D} = \left\lceil \frac{m}{V} \right\rceil \times n \times p \times \frac{D}{2} \tag{9}$$

Adding the above delays to the latency for processing B number of meshes (i.e. batch size) from [19] gives the total latency of a 2D and 3D application as equations (10) and (11) respectively:

$$Clks_{2D} = \frac{n_{iter}}{p} \times \left(\left\lceil \frac{m}{V} \right\rceil \times n \times B + delay_{2D}\right) \tag{10}$$

$$Clks_{3D} = \frac{n_{iter}}{p} \times \left(\left\lceil \frac{m}{V} \right\rceil \times n \times l \times B + delay_{3D}\right) \tag{11}$$

We make use of the models developed here to predict performance of the applications benchmarked in Section 5.

4 MULTI-DIMENSIONAL TRIDIAGONAL SOLVERS

FPGA synthesis of implicit solvers, poses similar but somewhat different challenges. Due to the order dependence when iterating over mesh elements, techniques such as the cell parallel method cannot be readily utilized. The best tridiagonal solver algorithms for implementing data-flow optimizations needs to be examined together with the utility of SYCL for their synthesis on Intel FPGAs.

We base our investigation and development on recent work [20], again done with Xilinx FPGAs. The work in [20] explores the various algorithms for implementing multi-dimensional tridiagonal solvers on FPGAs, focusing on gaining higher throughput from multiple solvers setup on a multi-dimensional domain. This aligns closely with how industrial workloads utilize tridiagonal solvers, particularly from the financial computing domain, as opposed to the best algorithms and optimizations for a single tridiagonal system solve as found in many other previous work. The key designs from [20] points to a synthesis based on the well known Thomas algorithm [36](see Alg. 1) as the best performing implementation. We follow the same design with SYCL. The Thomas algorithm 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 $O(N)$. Implementing Alg. 1 using floating-point primitive cores on

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 $i = 1, 2, ..., N-1$ 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 $i = N-2, ..., 1, 0$ do
9:     $d_i \leftarrow d_i^* - c_i^* d_{i+1}$
10: end for
11: return $d$
```

an Intel FPGA such as the PAC D5005 would incur an arithmetic pipeline latency of 37 clock cycles for the forward path and 6 clock cycles for the backward path. The forward path cycles are dominated by the slow floating-point division operations (26 clks latency for division operation). This essentially gives a dependency distance (l_f) of about 37 clks (taking the maximum out of the forward loop and backward loops latencies). Then, [20] demonstrates how $g = l_f$ number of tridiagonal systems can be grouped and interleaved such that iteration 1 of system 1 is input to the pipeline, followed by iteration 1 of system 2 and so on, per clock cycle, up to iteration 1 of system g. This allows to obtain higher throughput by continuously utilizing the computational pipeline verses solving one system at a time. Techniques such as double buffering (i.e. ping-pong buffers), can be used to further optimize the implementation. With SYCL, this can be codified with three kernels, one each for forward and backward loops and one for the interleaving of the systems.

With a minimum group size of $g_f = 37$, on the Intel PAC D5005, to solve systems with size N, with interleaving, the Thomas solver would require four on-chip block RAMs (for a, b, c, d) with $2 \times g_f \times N$ number of words, totaling $8 \times g_f \times N$ words. The 2× is due to the need of twice as much memory to setup ping-pong buffers. Storage for c^*, d^* for the forward pass kernel will require two RAMs with $2 \times g_f \times N$, totaling $4 \times g_f \times N$ words. Additionally storage for u in backward pass would require a RAM with $2 \times g_b \times N$ words. As such a total of $456 \times N$ words is required for the Thomas solver implementation on this specific FPGA. Additional RAMs with a smaller number of words will be required for storing the previous iteration values as detailed in [20].

In some applications, the coefficients a, b, c can be generated without reading from external memory, using an initialization routine. In these cases, coefficients a, b, c need not be interleaved, but instead could be calculated as part of the interleaving kernel. Fusing this coefficient generation would reduce the total memory cost to $234 \times N$ words. Further saving of on-chip memory can be done for the Intel FPGAs by separating the calculation of r to a separate kernel which we denote as a `r_generator` kernel. This can be done for Alg. 1 by creating a kernel with only line 4 and 6. Line 5 will be a separate kernel that will get the computed value of r through a pipe. Calculating r only requires coefficients a, b, c which again can be internally calculated within the `r_generator` kernel. Now, `r_generator` kernel would require group size $g_r = 37$, `Thomas_forward` and `interleave` kernels would require group size of $g_f = 9$. The group size of `Thomas_backward` remains same. This optimisation reduces the total memory cost to $140 \times N$ words. It is a 69% reduction from the non-fused version and 40% reduction compared to a fused but no `r_generator` variant in [20].

Table 1: Experimental systems specifications.

FPGA	Intel PAC D5005 [6]
DSP blocks	5760
MLABs / M20K	7.6MB / 29.3 MB
DDR4	64GB, 76.8GB/s, in 4 banks (1 channel/bank)
Host	Intel Xeon Platinum 8256 @3.8GHz
	(16 CPUs, 4 cores each)
	1559 GB RAM, Ubuntu 18.04.6 LTS
Design SW	Intel oneAPI 2021.4.0, Intel Quartus software 19.2
board_variant	pac_s10
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 Thomas solver can be vectorized to solve multiple systems in parallel. Again this can be done using a wider data path using arrays inside *struct* as illustrated in Section 3's `struct dPath16`. In our implementation, template parameters are used to specify the vectorization factor, data type, group size and input and output pipe index of the pipe array. We use a manually flattened loop with custom ping-pong buffers to save device resources using custom integer data types for loop controls and improve the latency by continuous execution than a nested loop based ping-pong buffer implementation. However manually flattened loops require a dependency distance that can be specified using the `[[intel::ivdep(safelen)]]` attribute.

A vectorized Thomas solver poses a challenge in feeding data depending on its layout. It is efficient to read using the 512-bit AXI bus on the FPGA to obtain high memory throughput. But this 512 bit/16 float(FP32) values will correspond to accessing the same tridiagonal system for when solving reasonably large systems. If the systems are organized one after another in memory, then to feed groups of different systems to the vectorized solvers requires additional units and transformation. Since reading sequential data in an AXI burst mode is efficient, we read 8 systems (FP32) and 4 systems (FP64) and internally buffer them. Then we read an 8×8 block from 8 different systems and an 8×8 transpose to get the values from different systems. These two kernels are implemented using ping-pong buffers for parallel execution. The implementation will require $2 \times 8 \times 8 \times 32 = 4096$ bits of registers. The design is identical to the one developed in [20] and the same performance models can be applied to predict performance of applications as demonstrated in the next section.

5 PERFORMANCE

We present experimental results of applying our design strategy by implementing two non-trivial, representative applications using SYCL on Intel FPGAs. The first application is a stencil application implementing an explicit numerical solver and the second application use multi-dimensional tridiagonal solvers. The applications are synthesized on an Intel PAC D5005. We use the non-USM (Unified Shared Memory) model of SYCL as it provides a simpler memory access implementation compared to the alternative USM model [32]. Finally, we compare the performance of the applications on the FPGA to equivalent implementations of the same applications, written in CUDA[2], on an Nvidia V100 GPU. Specification of FPGA and GPU Systems along with the specific software tools used in the evaluation are detailed in Table 1.

5.1 Reverse Time Migration(RTM) Forward-Pass

The Reverse Time Migration(RTM) forward-pass is a 3D application that uses a high-order stencil on vector elements. It consists of multiple stencil kernels inside an iterative time-marching loop. The high-level algorithm of RTM is detained in Alg. 2. Here K, T, Y, S are

Algorithm 2: RTM - Forward Pass

```
1: for $i = 0,\ i < n_{iter},\ i++$ do
2:     $K = f_{pml}(Y_{25pt}, \rho, \mu) \times dt;\ T = Y + K/2;\ S = K/6$
3:     $K = f_{pml}(T_{25pt}, \rho, \mu) \times dt;\ T = Y + K/2;\ S = S + K/3$
4:     $K = f_{pml}(T_{25pt}, \rho, \mu) \times dt;\ T = Y + K;\ S = S + K/3$
5:     $K = f_{pml}(T_{25pt}, \rho, \mu) \times dt;\ Y = Y + S + K/6$
6: end for
```

3D meshes with six floating-point elements at each mesh point. K, T are the intermediate meshes while Y is the initial mesh at the start of each iteration and S is a mesh that is the scaled sum of K. ρ, μ are 3D constant meshes with scalar elements. An 8^{th} order stencil with 25-points is used in the f_{pml} function. For synthesis on the FPGA, intermediate meshes such as K, T, S can be replaced by SYCL pipes due to their sequential element access pattern. Additionally, in order to reduce the global memory access for ρ, μ, Y, they are read once, then internally buffered to all kernels. Such buffering reduces the required off-chip memory bandwidth to be within the available 76.8 GB/s limit. An equivalent GPU implementation will require an additional mesh due to dependency in reading and writing of mesh T. With this, a GPU implementation will consist of four kernels in a iterative loop implemented on the Host.

Managing the on-chip memory to enable the execution of larger mesh sizes is a challenge for higher order stencil applications with vector mesh elements. We attempted to maximize the vectorization factor to reduce the iterative unroll factor to save on-chip memory while maintaining the same compute throughput. The maximum possible vectorization factor is 4, due to the off-chip memory bandwidth limitation. We opted to set this to 3, as it then allowed us to have an iterative loop unroll factor of 2 as well (a vectorization factor of 4 and an unroll factor of 2 will result in a design that will hit the upper limit of the available DSP units). Previous work in [19], which implements the RTM application on Xilinx's U280 FPGA, didn't attempt vectorization. This was due to the organization of the U280 into SLR portions, and a single SLR region not having sufficient DSP resources for such an implementation.

Application runtimes are detailed in Figure 7. Mesh sizes from 10^3 to 40^3 are executed with batch sizes (B) of 10 and 100. The model is also used to predict the runtime (dotted line noted as FPGA-Pred) for each case and prediction error is below 5%. It shows that the FPGA performance is on par or better compared to the V100 GPU performance. We attribute this to the availability of larger number of DSP units with native support for floating-point operations. Table 2 compares the effective bandwidth and energy consumption on the FPGA with the GPU. Bandwidth Utilisation is provided for GPU which mainly depend on global memory. FPGA's bandwidth utilisation is underpinned by the fast on-chip memory performance (with tens of TB/s) which we do not show here. As such, the FPGA's effective bandwidth reach up to 418GB/s even though global memory bandwidth is limited to 76.8 GB/s. Utilizing fast on-chip memory performance is a direct consequence of using window buffers and communication between stencil compute kernels via pipes. GPU reaches bandwidths of up to 681GB/s for the 16^3 utilizing higher portions of its peak bandwidth. This indicates near-optimal performance from the GPU implementation. We explored both Array of Structure (AoS) and Structure of Arrays (SoA) data layout for the vector elements on the GPU. SoA gives the best throughput due to better data access patterns. We speculate that the reduced performance for larger mesh sizes is due to poor cache utilization when solving higher order stencils.

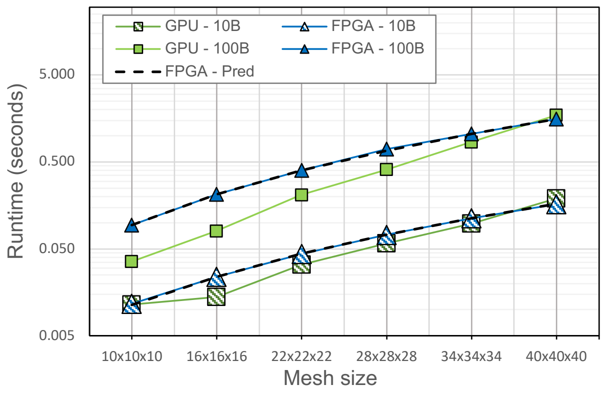

Figure 7: RTM forward-pass, FP32, $p = 2$, $v = 3$, 200 iterations

Table 2: RTM - Avg. Bandwidth, BW (GB/s), Avg. Utilisation(%) and Energy, E (J), 200 iters.

Mesh	BW-10B		BW-100B		E-100B	
	FPGA	GPU	FPGA	GPU	FPGA	GPU
10^3	154	158 (18%)	192	506 (56%)	8.5	4.8
16^3	230	391 (43%)	258	681 (76%)	19.4	12.9
22^3	286	379 (42%)	313	598 (66%)	36.2	37.7
28^3	331	414 (46%)	342	588 (65%)	62.9	75.2
34^3	367	420 (47%)	390	486 (54%)	96.2	164.6
40^3	397	344 (38%)	418	379 (52%)	141.3	344.7

We used the `fpgainfo` utility to measure power consumption on the PAC D5005. The utility gives voltages and current of the 12V PCIe power supply as well as 12 V AUX power supply. GPU power consumption is obtained through `nvidia-smi`. For RTM, power consumption of the Intel PAC D5005 is between 84-94W while the V100 GPU's power consumption is between 47-200W. Observed power-draw indicate that the FPGA is just over 59% less energy consuming than the GPU for the largest mesh with the larger batch sizes.

5.2 ADI 2D Heat Diffusion Application

The Alternating Direction Implicit (ADI) time discretization requires multiple tridiagonal systems solved in multiple dimensions.

Our second application implements the 2D heat diffusion equation using ADI. The High-Level algorithm is detailed in Alg. 3. The application iterates for a given number of iterations and in each iteration the RHS values for a tridiagonal matrix coefficients are produced using a 2D stencil loop. Then, tridiagonal systems are solved along the x-dimension, *Tridslv(x-dim)* and along the y-dimension, *Tridslv(y-dim)*. Finally, solution d is accumulated on to the with initial value of u. An equivalent GPU implementation consists of three kernels for each of the above. We use the batched `tridsolver` library developed in [22] in this evaluation but note that using Nvidia's CuSparse library also resulted in similar performance.

Algorithm 3: 2D ADI Heat Diffusion 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:    $u = u + d$
6: end for
```

Performance on FPGAs could be maximized by pipelining all four steps in Alg. 3 due limitations in global memory bandwidth. Intermediate results from *Tridslv(x-dim)* will need to be transposed using on-chip memory to achieve this. Once all the kernels are pipelined, the iterative loop can also be unrolled. The implementation in [20], on the Xilinx U280 for the same application, used an unroll factor of 3 and then scaled the design to multiple compute units (CUs) based on available HBM ports. Scaling to multiple CUs, instead of using a higher unroll factor results in lower latency for small batch sizes. Since HBM memory is not available on the Intel D5005, we preferred to unroll the iterative loop instead of scaling to CUs in the present work. The performance model for this implementation including the scheduling latency (due to the existence of a stencil loop) can be noted as in equation (12):

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

$$L_{rhs+xy} = f_U \times [(2x/v) + (2vx/v + 3gx) + (2xy/v + 3gy)] + B \times (xy/v) + \sum_{i=0}^{kernels} (clk_{wr,i} - clk_{rd,i}) \tag{13}$$

Here, x, y are mesh sizes, B is the batch size, v is the vectorization factor, g is the group size of systems, f_U is the unroll factor of the iterative loop. This is similar to the models created for the 2D ADI application in [20].

Figure 8 gives runtime performance of 2D ADI Heat Diffusion application in FP32 on the Intel PAC D5005 and compares it to performance on the Nvidia V100 GPU. Even though the FPGA implementation operates at 231MHz, it outperforms the GPU. We attribute this to the unrolling of the iterative loop by a factor of 8 and the fusion of coefficient in the Thomas solver. This essentially allows data to be kept on faster on-chip memory without writing to global (external) memory for the whole computation. The same type of fusion is not supported by the GPU implementation as the GPU tridiagonal solver [2] call is a function call to an external library. Additionally it does not support fusion of coefficient generation internally.

We can estimate the runtime of the GPU if coefficients were generated internally, assuming that the GPU is not compute limited

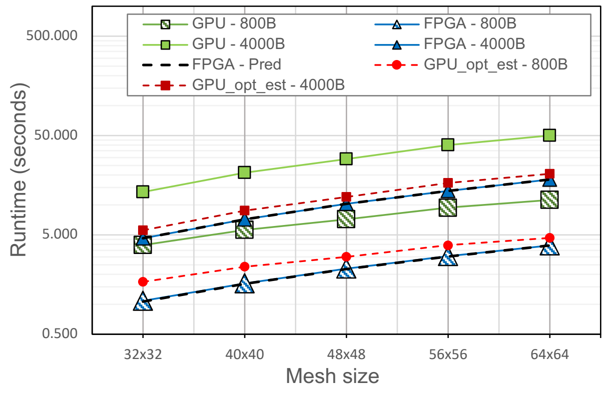

Figure 8: ADI 2D, FP32, $f_U = 8, V = 8$, 16000 iter meshes

and the same sustained bandwidth is maintained for each of the application cases. Estimated Run times for each kernel call can be computed using equation (14). Here, t_{opt}, t are run-times for the GPU implementations with and without internally generated coefficients respectively. When generating coefficients internally, data movement does not include data structures coefficient meshes a, b, c. Then the run-time estimate for the full ADI application is can be obtained from equation (15), where runtimes are adjusted for the *RHS* calculation, *Tridslv(x-dim)* and *Tridslv(y-dim)*. Here, we note that the accumulation step in Alg. 3 is fused into *Tridslv(y-dim)* of the GPU implementation.

$$t_{opt} = t \times data\_movement_{opt}/data\_movement \tag{14}$$

$$t\_adi_{opt} = \frac{1}{4} \times t_{preproc} + \frac{2}{5} \times t_{xsolve} + \frac{4}{7} \times t_{ysolve} \tag{15}$$

Even when the coefficients are internally generated on the GPU, the FPGA appear to perform marginally better, as can be see by the dotted red lines in Figure 8. The model predicted runtimes for FPGA is closely matching with the actual runtimes with a prediction error of less than 2%.

Table 3 details the effective bandwidth of the FPGA and GPUs with bandwidth utilisation for GPU and Energy consumption for both devices. The GPU bandwidth is noted for x and y solves separately given that these are separate calls to the tridiagonal solver library. The FPGA bandwidth reaches up to 463GB/s. This, as noted above, is due to on-chip memory based data movement without reading/writing from lower bandwidth global memory. In GPU, *Tridslv(y-dim)* reaches a good bandwidth of 555 GB/s but *Tridslv(x-dim)* performs poorly, only reaching up to 205 GB/s. Such lower bandwidths are also reported by [20, 22] due to the 8×8 transpose operations using registers/shared memory on GPUs. FPGA power consumption varies between $95 - 101$W while the GPU power consumption varies between $105 - 151$W. for the largest mesh with running the largest batch size, the FPGA saves over 76% energy used compared to the GPU. The same application on the U280 [20] used HBM based delay buffers[12] to save on-chip memory and managed to run mesh sizes of up to 128×128. On the Intel PAC D5005, larger delay buffers are also implemented using on-chip memory and this limits the largest mesh size to 64×64.

Table 3: ADI Heat Diffusion Application: Achieved Bandwidth, BW (GB/s) Utilisation(%) and Energy, E (KJ)

2D FP32 (16000 iterations, f_U = 8), F - FPGA, G - GPU								
Mesh	BW-800B			BW-4000B			E-4000B	
	F	Gx	Gy	F	Gx	Gy	F	G
32^2	386	131 (15%)	288 (32%)	453	185 (21%)	524 (58%)	0.453	1.825
40^2	403	144 (16%)	330 (37%)	457	197 (22%)	478 (53%)	0.712	3.159
48^2	414	163 (18%)	389 (43%)	460	202 (22%)	517 (57%)	1.032	4.363
56^2	422	169 (19%)	412 (46%)	462	202 (22%)	498 (55%)	1.411	6.220
64^2	428	182 (20%)	477 (53%)	463	205 (23%)	555 (62%)	1.820	7.580

6 CONCLUSION

In this paper, we explored the design and development of structured-mesh based solvers using SYCL for Intel FPGA hardware. Two classes of applications were targeted (1) stencil applications based on explicit iterative methods and (2) multi-dimensional tridiagonal solvers based on implicit methods. A generalized workflow, extending previous work in [19] and [20], for synthesizing optimized solvers of these applications was developed together with an analytic model to predict their performance in support of design space explorations. The extensions targeted key optimizations required to obtain the best performance using SYCL programming techniques. The main methods for codification with SYCL include (1) reducing SYCL kernel calling overhead by moving time-marching outer loop on to the FPGA device and (2) reducing on-chip memory usage for the Thomas solver implementing multi-dimensional tridiagonal solvers. The designs and workflow was applied to two non-trivial applications synthesizing them on an Intel PAC D5005 FPGA. Performance results were compared to the same applications implemented on an Nvidia V100 GPU as a baseline. Observed results indicate the FPGA provided better or matching performance compared to the V100 GPU in terms of runtime. We also see 59%-76% less power consumption when executing these applications at its largest mesh and batch sizes. The performance models provided high accuracy with less than 5% model prediction errors for all cases. Future work will extend these techniques to other Intel FPGAs with HBM memory and also consider applications with larger meshes that was currently limited by the PAC D5005's hardware resources. We will also compare the performance of our multi-dimensional tridiagonal solver design to Intel's tridiagonal solver library.

The significant effort in applying non-trivial transformation to optimize the SYCL implementations demonstrate the programming overheads still dominating the development of programs to utilize FPGAs. This is still true even with the hardware vendor's providing mature HLS tools for development. While we have not quantified the productivity overheads in this paper, it is clear that such hand-tuned, hardware specific programming is not tractable particularly for developing and maintaining codes for execution on multiple hardware such as GPUs and FPGAs, even with language extensions such as SYCL. Future work will aim to utilize domain specific languages and automatic code-generation techniques, such as OPS [31] and OP2 [26] to target FPGAs using the optimizations and transformations developed in this paper.

The code developed as part of this work is available as open-source software at [4] and [5].

7 ACKNOWLEDGMENTS

Gihan Mudalige was supported by the Royal Society Industry Fellowship Scheme (INF/R1/1800 12). We are grateful to Intel for providing access to the PAC D5005 FPGA via their DevCloud platform.

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 Quantitative Finance Library V.2020.2. https://xilinx.github.io/Vitis_Libraries/quantitative_finance/2020.2/.
[4] 2022. High-Level FPGA accelerator design for structured-mesh-based explicit numerical solvers - GitHub Code Repository. https://github.com/Kamalavasan/StencilsOnFPGA.
[5] 2022. High Throughput Multidimensional Tridiagonal System Solvers on FPGAs - GitHub Code Repository. https://github.com/Kamalavasan/Tridsolver-FPGA.
[6] 2022. Intel® FPGA Programmable Acceleration Card D5005. https://www.intel.com/content/www/us/en/products/details/fpga/platforms/pac/d5005.html.
[7] 2022. Pipe Array - GitHub Code Repository. https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming/DPC%2B%2BFPGA/Tutorials/DesignPatterns/pipe_array.
[8] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. 2009. A view of the parallel computing landscape. *Commun. ACM* 52, 10 (Oct 2009), 56–67. https://doi.org/10.1145/1562764.1562783
[9] Tobias Becker, Oskar Mencer, Stephen Weston, and Georgi Gaydadjiev. 2015. Maxeler data-flow in computational finance. In *FPGA Based Accelerators for Financial Applications.* 243–266.
[10] Y. Chi, J. Cong, P. Wei, and P. Zhou. 2018. SODA: Stencil with optimized dataflow architecture. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD).*
[11] Johannes de Fine Licht, Maciej Besta, Simon Meierhans, and Torsten Hoefler. 2021. Transformations of High-Level Synthesis Codes for High-Performance Computing. *IEEE Trans. Parallel Distrib. Syst.* 32, 5 (may 2021), 1014–1029. https://doi.org/10.1109/TPDS.2020.3039409
[12] 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.* IEEE Press, 315–326. https://doi.org/10.1109/CGO51591.2021.9370315
[13] Johannes de Fine Licht, Andreas Kuster, Tiziano De Matteis, Tal Ben-Nun, Dominic Hofer, and Torsten Hoefler. 2020. StencilFlow: mapping large stencil programs to distributed spatial computing systems. https://arxiv.org/abs/2010.15218 CoRR arXiv:2010.15218.
[14] Jim Douglas and James E Gunn. 1964. A General Formulation of Alternating Direction Methods. *Numèrische mathèmatik* 6, 1 (1964), 428–453.
[15] Haohuan Fu and Robert G. Clapp. 2011. Eliminating the memory bottleneck: An FPGA-based solution for 3D reverse time migration. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays.* 65–74.
[16] Walter Gander and Gene H Golub. 1997. Cyclic Reduction—History and Applications. *Scientific computing (Hong Kong, 1997)* 7385 (1997).
[17] Mike Hutton. 2022. Understanding How the New Intel® HyperFlex™ FPGA Architecture Enables Next-Generation High-Performance Systems. (2022). https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01231-understanding-how-hyperflex-architecture-enables-high-performance-systems.pdf/.
[18] Q. Jia and H. Zhou. 2016. Tuning stencil codes in OpenCL for FPGAs. In *2016 IEEE 34th International Conference on Computer Design (ICCD).* 249–256.
[19] 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 *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* 1087–1096. https://doi.org/10.1109/IPDPS49936.2021.00117
[20] Kamalavasan Kamalakkannan, Istvan Z Reguly, Suhaib A Fahmy, and Gihan R Mudalige. 2022. High Throughput Multidimensional Tridiagonal Systems Solvers on FPGAs. *arXiv preprint arXiv:2201.03950* (2022).
[21] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* 242–251.
[22] 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.

[23] 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 *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1977–1980. https://doi.org/10.1109/ISCAS.2015.7169062

[24] 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. *Int. J. Reconfigurable Comput.* 2019 (2019), 3679839:1–3679839:13.

[25] 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 17th Biennial Computational Techniques and Applications Conference, CTAC-2014 (ANZIAM J., Vol. 56)*, Jason Sharples and Judith Bunder (Eds.). C446–C462. http://journal.austms.org.au/ojs/index.php/ANZIAMJ/article/view/9371

[26] Gihan R. Mudalige, Mike B. Giles, Istvan Z. Reguly, Carlo Bertolli, and Paul H. J. Kelly. 2012. OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. *Proceedings of Innovative Parallel Computing (InPar)* (2012).

[27] Tan Nguyen, Samuel Williams, Marco Siracusa, Colin MacLean, Douglas Doerfler, and Nicholas J. Wright. 2020. The Performance and Energy Efficiency Potential of FPGAs in Scientific Computing. In *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 8–19. https://doi.org/10.1109/PMBS51919.2020.00007

[28] 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.

[29] Eric Polizzi and Ahmed H. Sameh. 2006. A Parallel Hybrid Banded System Solver: the SPIKE Algorithm. *Parallel Comput.* 32, 2 (2006), 177–194. https://doi.org/10.1016/j.parco.2005.07.005 Parallel Matrix Algorithms and Applications (PMAA'04).

[30] Prashant Rawat, Martin Kong, Tom Henretty, Justin Holewinski, Kevin Stock, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2015. SDSLc: A multi-target domain-specific compiler for stencil computations. In *Proceedings of the International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*. Article 6, 10 pages.

[31] I. Z. Reguly, G. R. Mudalige, and M. B. Giles. 2018. Loop tiling in large-scale stencil codes at run-time with OPS. *IEEE Transactions on Parallel and Distributed Systems* 29, 4 (April 2018), 873–886. https://doi.org/10.1109/TPDS.2017.2778161

[32] James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, and Xinmin Tian. 2021. *Unified Shared Memory*. Apress, Berkeley, CA, 149–171. https://doi.org/10.1007/978-1-4842-5574-2_6

[33] K. Sano, Y. Hatsuda, and S. Yamamoto. 2011. Scalable streaming-array of simple soft-processors for stencil computations with constant memory-bandwidth. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*. 234–241.

[34] M. Schmidt, M. Reichenbach, and D. Fey. 2012. A generic VHDL template for 2D stencil code applications on FPGAs. In *Proceedings of the IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*. 180–187.

[35] M. Shafiq, M. Pericàs, R. de la Cruz, M. Araya-Polo, N. Navarro, and E. Ayguadé. 2009. Exploiting memory customization in FPGA for 3D stencil computations. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*. 38–45.

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

[37] H. M. Waidyasooriya and M. Hariyama. 2019. Multi-FPGA accelerator architecture for stencil computation exploiting spacial and temporal scalability. *IEEE Access* 7 (2019), 53188–53201.

[38] H. M. Waidyasooriya, Y. Takei, S. Tatsumi, and M. Hariyama. 2017. OpenCL-based FPGA-platform for stencil computation and its optimization methodology. *IEEE Transactions on Parallel and Distributed Systems* 28, 5 (2017), 1390–1402.

[39] 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. https://doi.org/10.1016/j.procs.2014.05.009 2014 International Conference on Computational Science.

[40] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. 2018. Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 153–162.

[41] H. R. Zohouri, A. Podobas, and S. Matsuoka. 2018. High-performance high-order stencil computation on FPGAs using OpenCL. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 123–130.

,