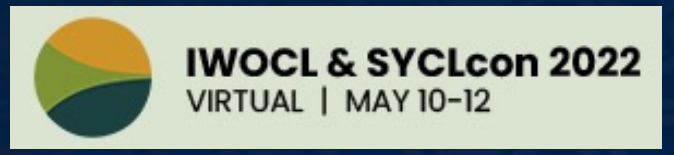
FPGA Acceleration of Structured-Mesh-Based Explicit and Implicit Numerical Solvers using SYCL

Kamalavasan Kamalakkannan, Gihan R. Mudalige
University of Warwick, UK

Istvan Z. Reguly
Pazmany Peter Catholic University, Hungary

Suhaib A. Fahmy
King Abdullah University of Science and Technology, Saudi Arabia

Email: kamalavasan.kamalakkannan@warwick.ac.uk

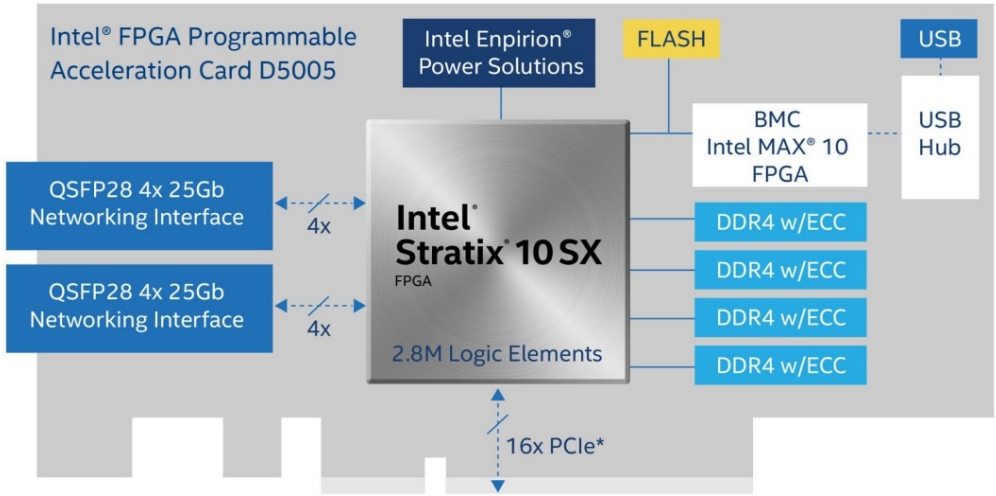


WARWICK
THE UNIVERSITY OF WARWICK

WHY FPGAS FOR HPC APPLICATIONS

- Field Programmable Gate Arrays (FPGAs) gaining traction as accelerator devices competitive to traditional architectures

  Examples :
  - Deep Neural Networks (DNN) – Binq, Baidu
  - Financial computing
  - Databases

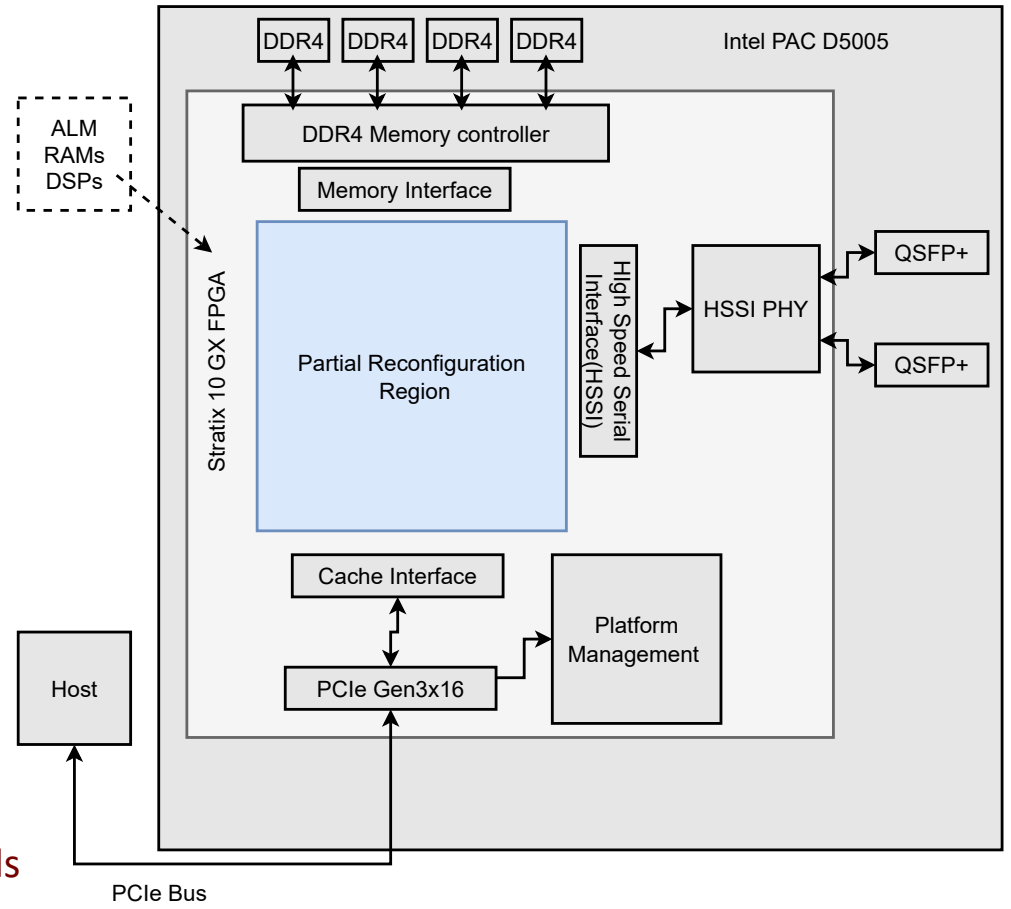


- Attractive features of FPGAs
  - High Performance for parallel algorithms – data-flow model
  - Energy efficient
  - Low latency
  - Reconfigurability – Software Defined Accelerator (SDA)

- FPGA programming made easier than before with powerful tools and hardware capabilities
  - Introduction of High-Level Synthesis (HLS) tools
  - HBM memory
  - Hardware single precession cores
  - Cloud FPGA node instances

FPGA Device and Programming Challenge

- FPGA device consists of configurable elements to implement an algorithm or application in a digital circuit
  - Adaptive Logic Modules(ALMs) – Look Up Tables(LUTs) and Registers
  - Random Access Memory Blocks – 640-bit MLABs and 20K bit M20K
  - Digital Signal Processing Blocks(DSPs)
  - Routing Fabrics to connect circuit elements
  - Hardened blocks – Memory Controller, PCIe , Clock Modules
- FPGA accelerator card additionally consists of big DDR4 memories, Network interfaces and PCIe interface to communicate with Host
- Challenge is generating optimal circuit for an application using a high-level language and requiring low level customization
  - User has to design memory hierarchy
  - Balance throughput and device resource consumption across many kernels
  - Design should consider larger reconfiguration time opposed to GPU kernel calls
  - Evolving HLS tools and limited debugging facilities



CONTRIBUTIONS

- Extending previous work for synthesizing structured-mesh solvers on Xilinx FPGAs
  - **Kamalakkannan et. al IPDPS2021** - High-Level FPGA Accelerator Design for Structured-Mesh-Based Explicit Numerical Solvers
  - **Kamalakkannan et. al ICS2022** - High Throughput Multidimensional Tridiagonal System Solvers on FPGAs

- In this work, we develop workflow to target same class of applications on **Intel FPGAs** using **SYCL**

- Codify design using SYCL by Overcoming challenges for gaining near-optimal performance :
  - Reducing kernel call overhead by moving iterative loop to FPGA
  - On chip memory saving for Thomas solver implementation using decoupled computation of forward loop

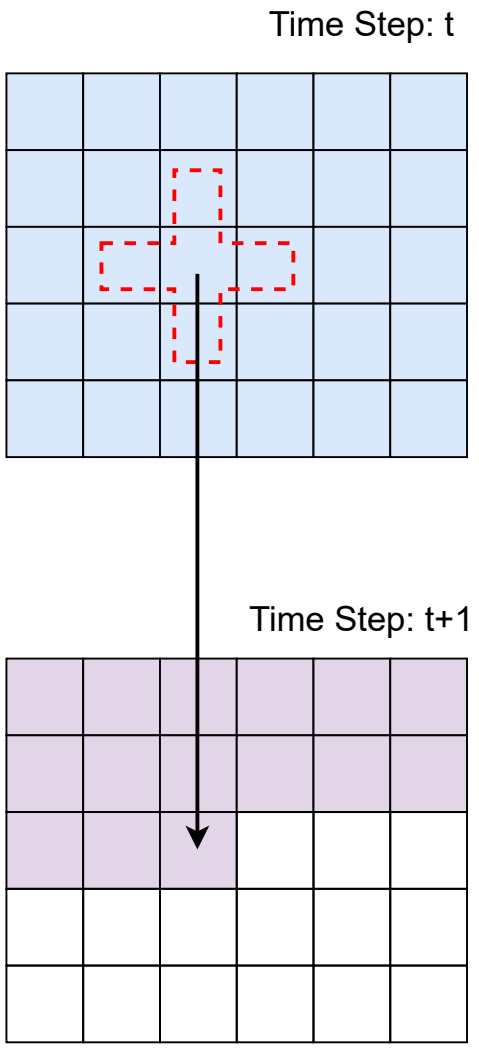
- Use design workflow for implementing two representative applications on Intel D5005 FPGA
  - Showcase use of SYCL to implement nontrivial application on FPGAs
  - Performance comparison with Nvidia V100 - compare best implementation on both architectures !
  - Competitive performance and significant energy saving is achieved compared to GPUs

Outline

- ❑ FPGAs for HPC – Motivation and challenges ✓
- ❑ Contributions ✓
- ❑ Application Class 1 - Stencil Solvers
  - Primitive Design
  - Challenges and Optimizations when using SYCL
  - [Performance Models]
- ❑ Application Class 2 - Multidimensional Tridiagonal System Solvers
  - Thomas Solver
  - Challenges and Optimizations when using SYCL - on chip memory saving optimization
  - [Performance models]
- ❑ Performance
  - Runtime and Bandwidth
  - Energy consumption
- ❑ Lessons learnt and Conclusions

Application Calss 1 - Structured Mesh based explicit stencil Solvers

```
for t in range (niter) {
  for x in range (height) {
    for y in range (width) {
      U^{t+1}_{x,y} = k1×U^t_{x-1,y} + k2×U^t_{x,y-1} + k3×U^t_{x+1,y} + k4× U^t_{x,y+1} + k5×U^t_{x,y}
    }
  }
}
```



- Finite Difference Methods(FDM) used to solve PDEs numerically,
- Stencils used to specify required points

- Naturally parallel – all cells could be updated in parallel

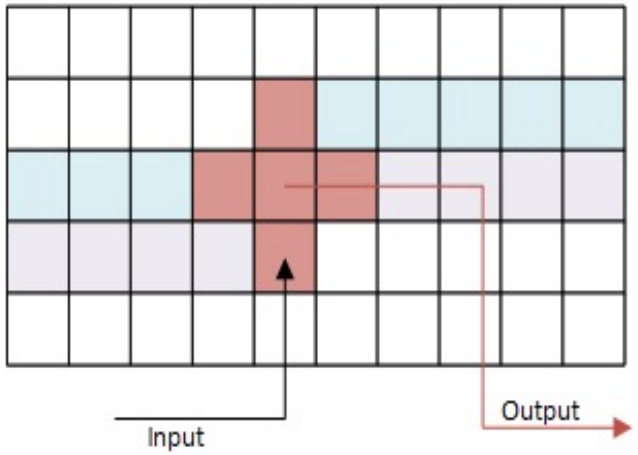
SYCL: Work Group based stencil implementation

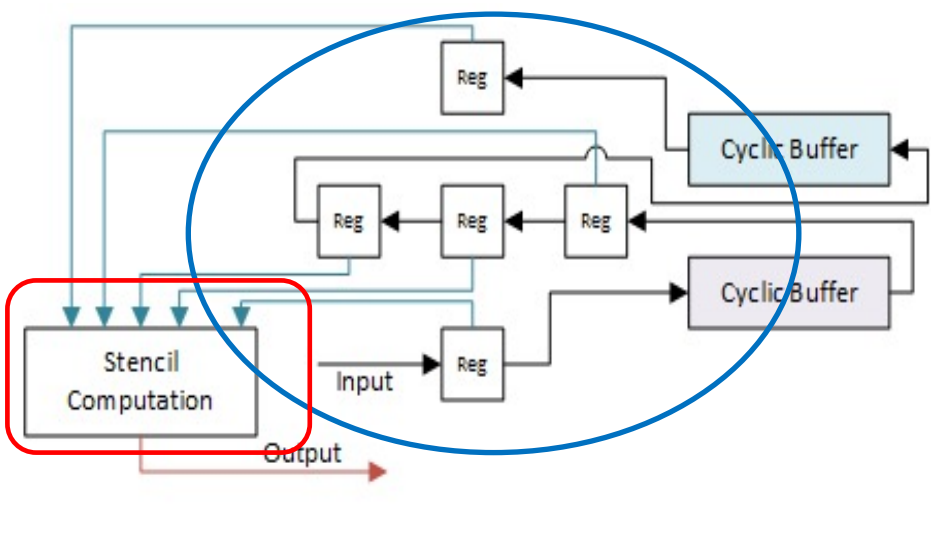
```
1  using namespace sycl;
2  void stencil_WI( queue &q,
3                   buffer<float,2> b_data_in,
4                   buffer<float,2> b_data_out,
5                   int size0, int size1,
6                   int block0, int block1){
7    q.submit([&] (handler& h){
8      accessor in(b_data_in, h);
9      accessor out(b_data_out, h);
10
11     range<2> local_range(block0, block1);
12     range<2> global_range(size0, size1);
13
14     h.parallel_for<class stencil_WI>
15     (nd_range<2>(local_range, global_range),
16     [=] (nd_item<2> point){
17       int y = point.get_global_id(0);
18       int x = point.get_global_id(1);
19       if(x > 0 && y > 0 && x < size0-1 && y < size1-1){
20         float r = (in[y-1][x] + in[y+1][x])*0.125f +
21           in[y][x]*0.5f;
22         out[y][x] = r;
23       }
24   });});
25 }
```

- Pros
  - Familiar GPU implementation style
- Cons
  - Performance depends on lower global memory bandwidth on FPGAs
  - Multiple memory access => multi-port cache
  - Required cache size unknown at compile time
  - Kernel to Kernel communication using pipes is not possible
    [more on this later !]

SYCL: Single task Implementation of stencil solver

```
1  using namespace sycl;
2  void stencil_ST(queue &q,
3                  buffer<float,2> &b_data_in,
4                  buffer<float,2> &b_data_out,
5                  int size0, int size1){
6    q.submit([&] (handler& h){
7      accessor  in(b_data_in, h);
8      accessor  out(b_data_out, h);
9      h.single_task<class stencil_ST> ([=] (){
10       float window1[MAX_DIM];
11       float window2[MAX_DIM];
12       float s_1_0, s_0_1, s_1_1, s_2_1, s_1_2;
13       /* +1 due to one row delay through window buffer */
14       int total_itr = (size1+1)*size0;
15       for(int i = 0; i < total_itr; i++){
16         int x = itr % size0;
17         int y = itr / size0;
18         int ptr = itr % (size0-1);
19
20         s_1_0 = window2[ptr];
21         s_0_1 = s_1_1;
22         window2[ptr] = s_1_1;
23         s_1_1 = s_2_1;
24         s_2_1 = window1[ptr];
25         if(y < size1) s_1_2 = in[y][x];
26         window1[ptr] = s_1_2;
27         float r = (s_1_0 +s_0_1 + s_2_1+ s_1_2)*0.125f +
28         s_11*0.5f;
29         if(x > 0 && y > 0 &&
30             x < size0-1 && y < size1){
31             out[y-1][x] = r;
32         }
33       }
34   });});
35 }
```

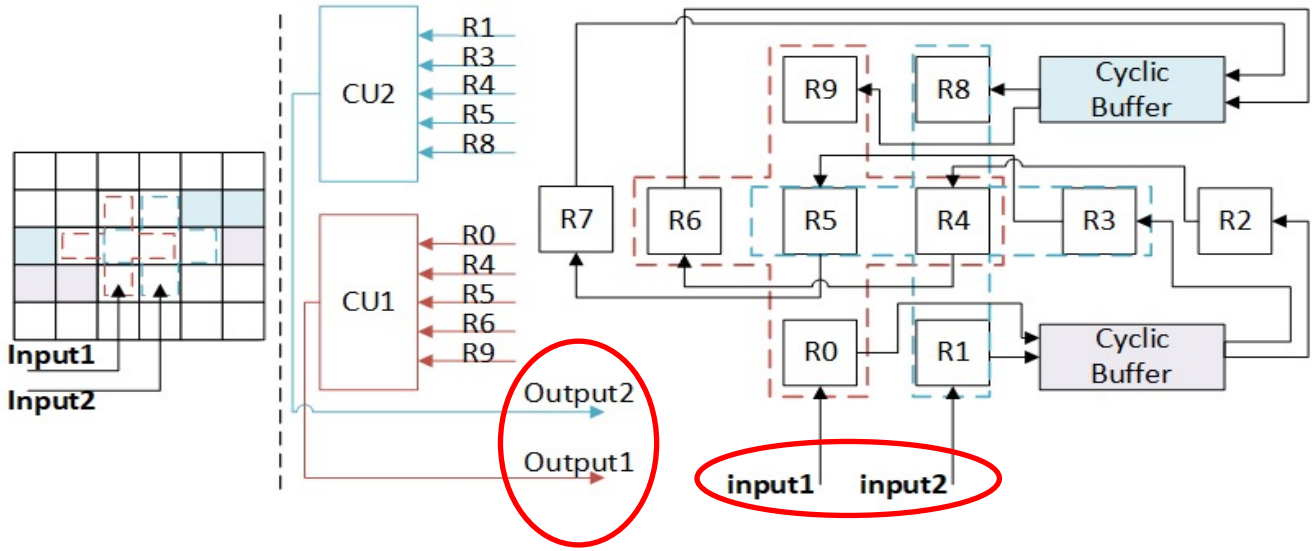


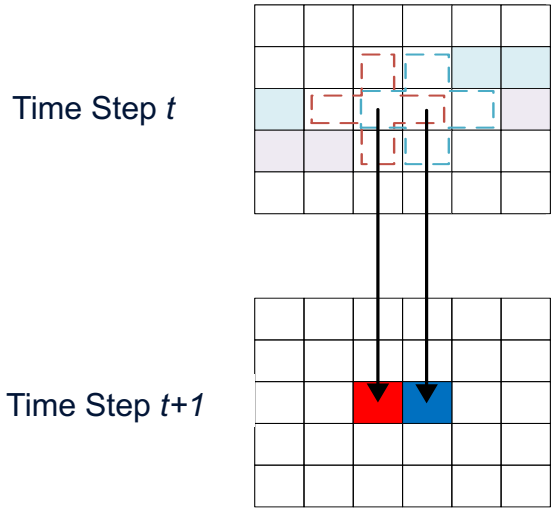


SYCL: VECTORIZATION

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

struct dPathV window1[MAX_DIM/VFACTOR];
struct dPathV window2[MAX_DIM/VFACTOR];
struct dPathV s_1_0, s_0_1, s_1_1, s_2_1, s_1_2, vec_wr;
for(int i = 0; i < total_itr; i++){
 /* other declarations, index calculation, window buffer*/
 float mid_1[VFACTOR+2] = {s_0_1.data[VFACTOR-1], \
   s_1_1.data[0], .. , s_1_1.data[VFACTOR-1] , s_2_1.data[0]};
 #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] + mid_1[v] + \
  mid_1[v+2])*0.125f+  s_1_1.data[v]*0.5f;
  bool cond = (i_ind>0 && i_ind<size0-1 && j>1 && j<size1);
  vec_wr.data[v]= cond ? val : s_1_1.data[v];
  }
  /* writing results to pipe */
}
```

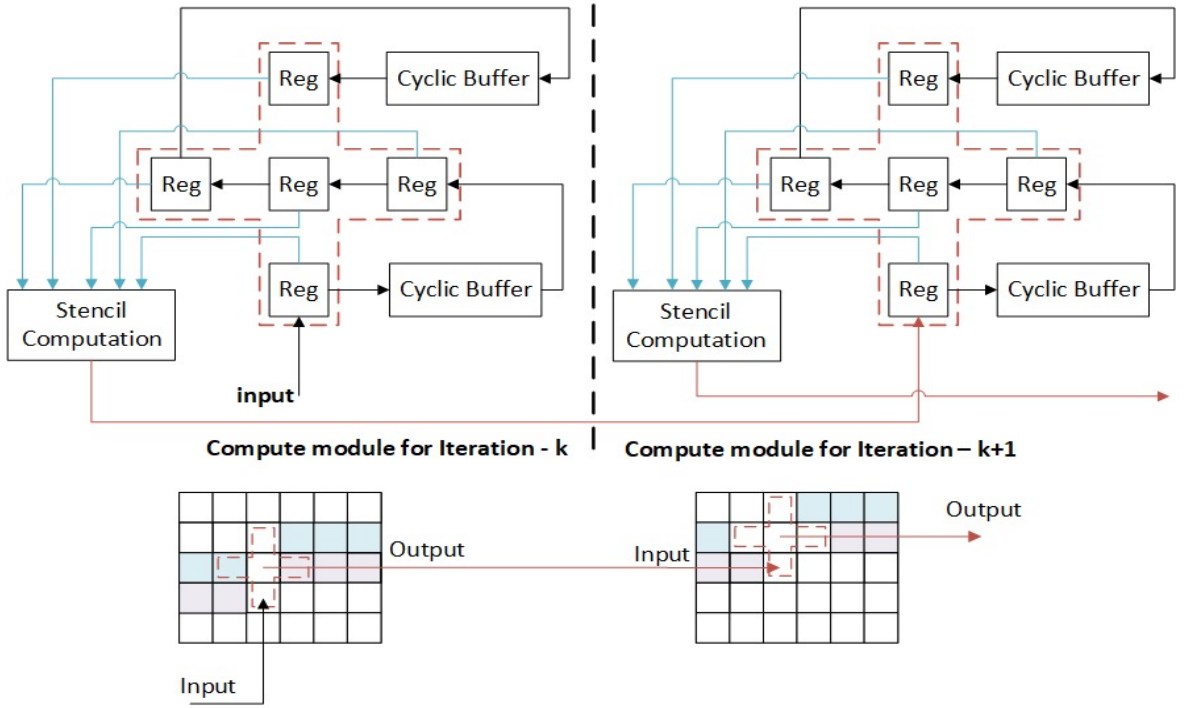




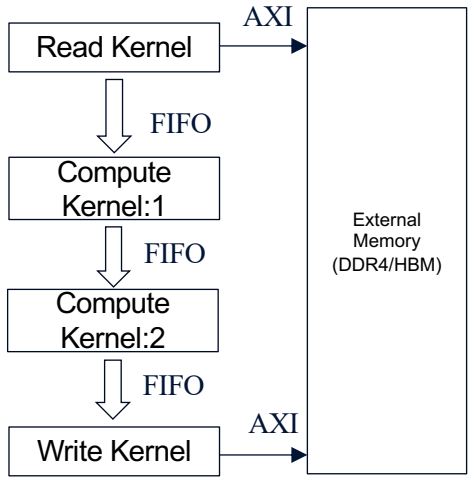
- Vectorization Factor: Number of mesh points updated at same clock
  - Called the "Cell parallel" method in Waidyasooriya et al. 2017
- Demands bandwidth proportional to vectorization factor
- On-chip memory requirement ideally same as primitive design

SYCL: Iterative loop Unrolling

```
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);
  }
};
```

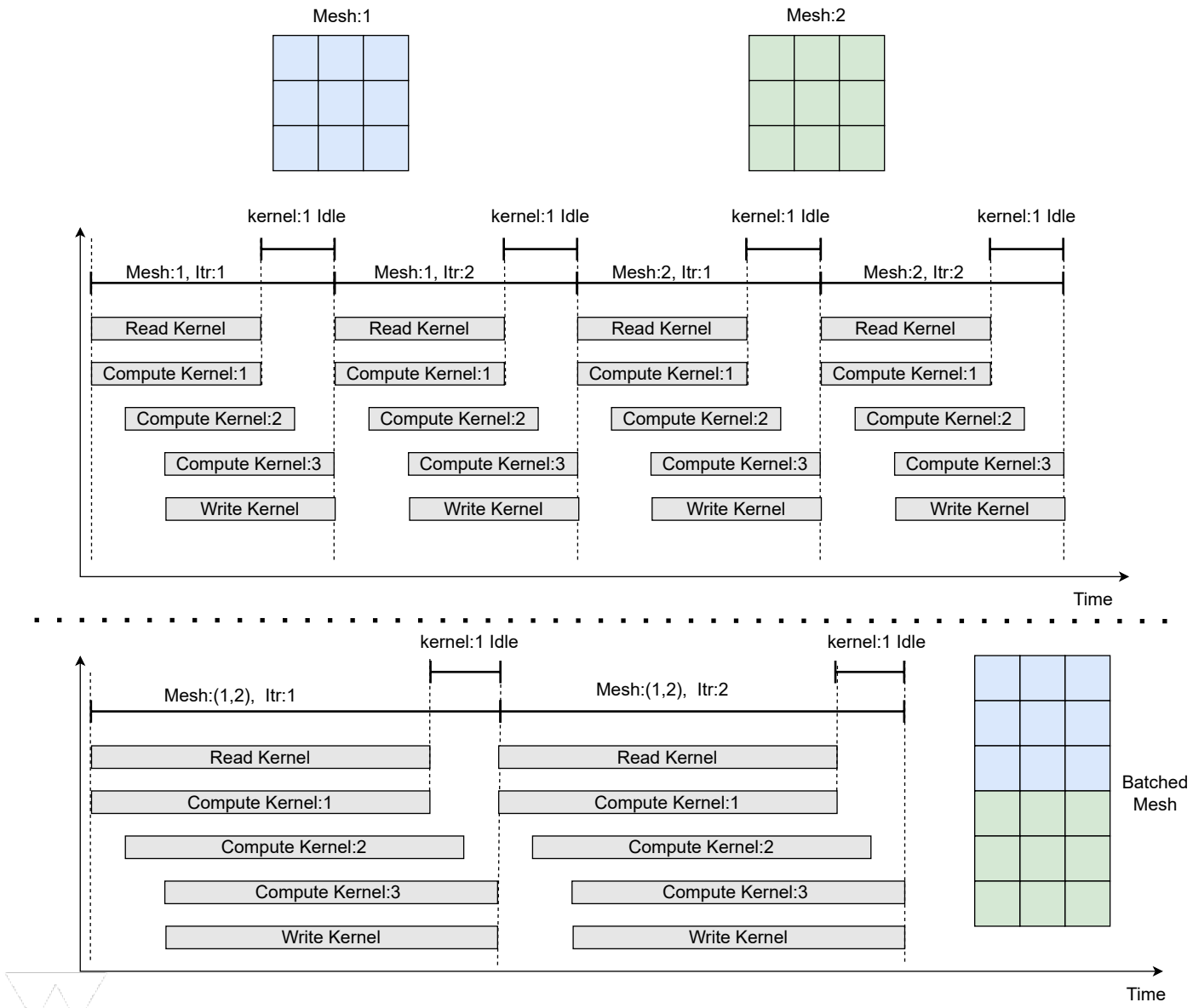


- Multiple Iterations in Parallel
  - Called as “Step parallel” method in Waidyasooriya et al. 2017
- Does not cost external memory Bandwidth
- On-chip memory ∝ unroll factor

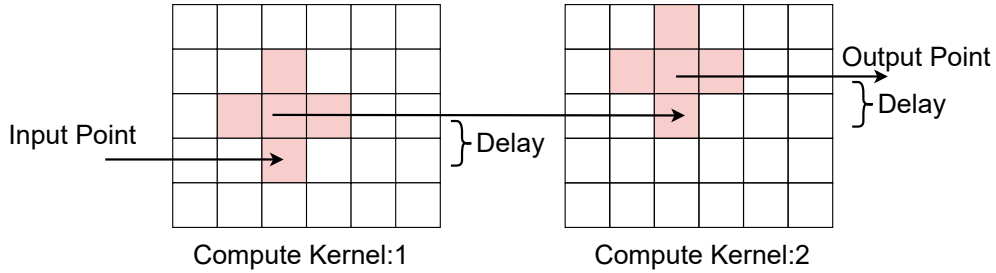


**Previous works utilized this technique - Waidyasooriya2017, soda2018*

SYCL: Batching



- Pipeline latency becomes significant for smaller grids

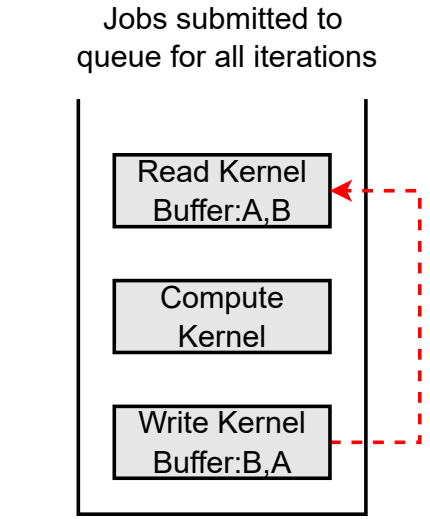


- Batching on last dimension
- Reduce latency per mesh
- No SYCL specific constructs required for batched processing

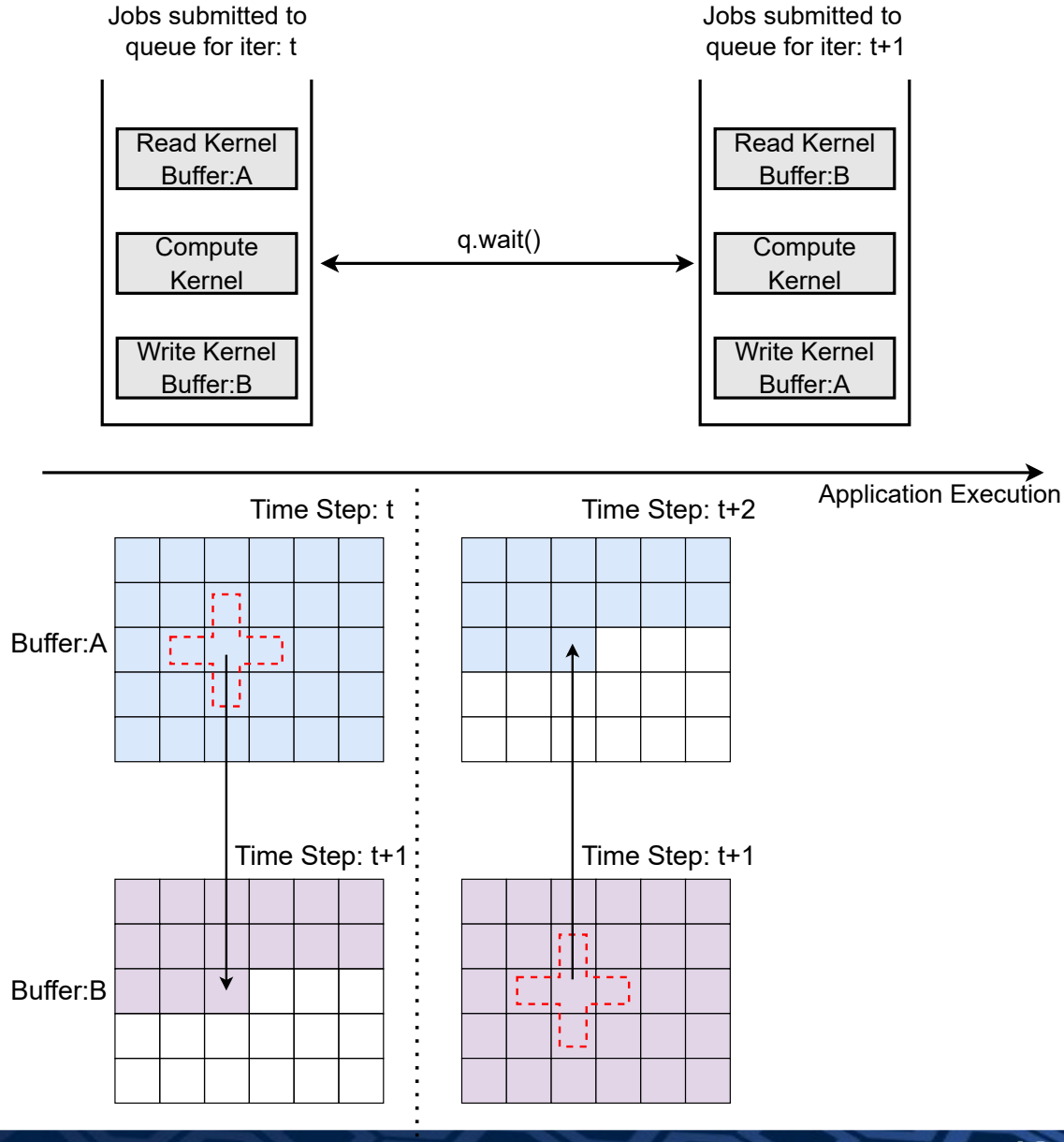
SYCL: Moving Iterative Loop to FPGA

- Why do we need to move Iterative loop to FPGA?
  - Kernel job submission overhead is significant for smaller meshes
  - Explicit memory access synchronization using `q.wait()` on host

- Moving Iterative loop to each FPGA kernel
  - Read and write kernel requires both buffers A,B
  - Requires a memory synchronization flag
  - Deadlock due to dependency-based runtime scheduling



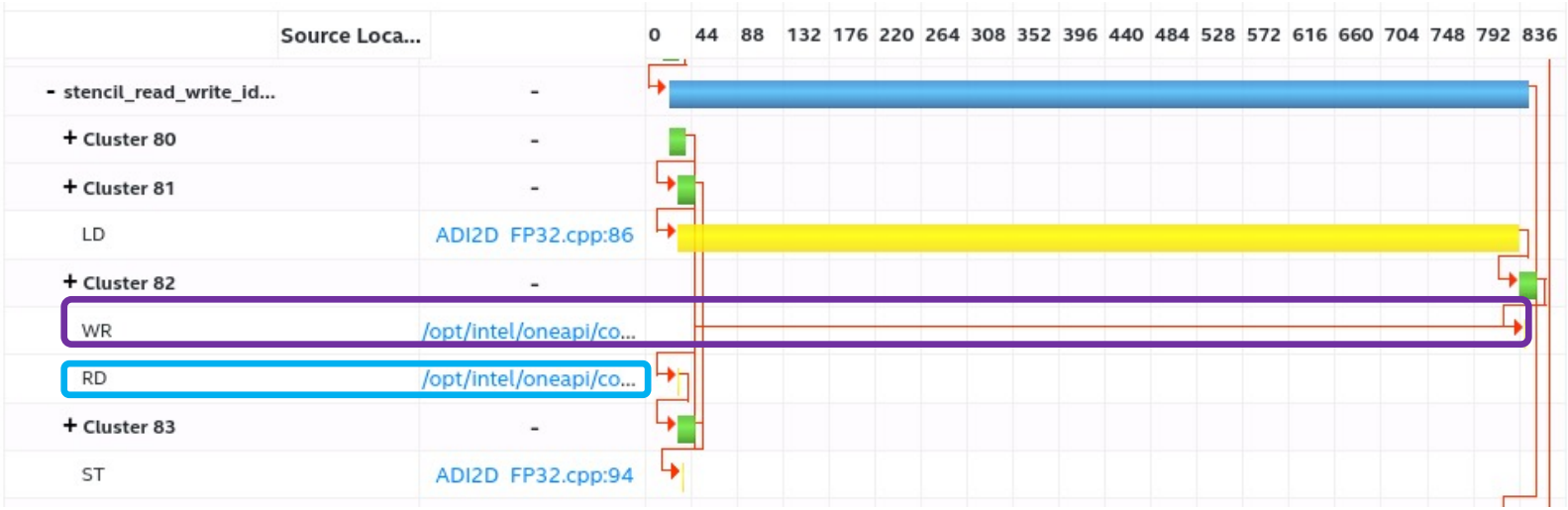
Runtime Identify Read after Write dependency for Buffers A,B



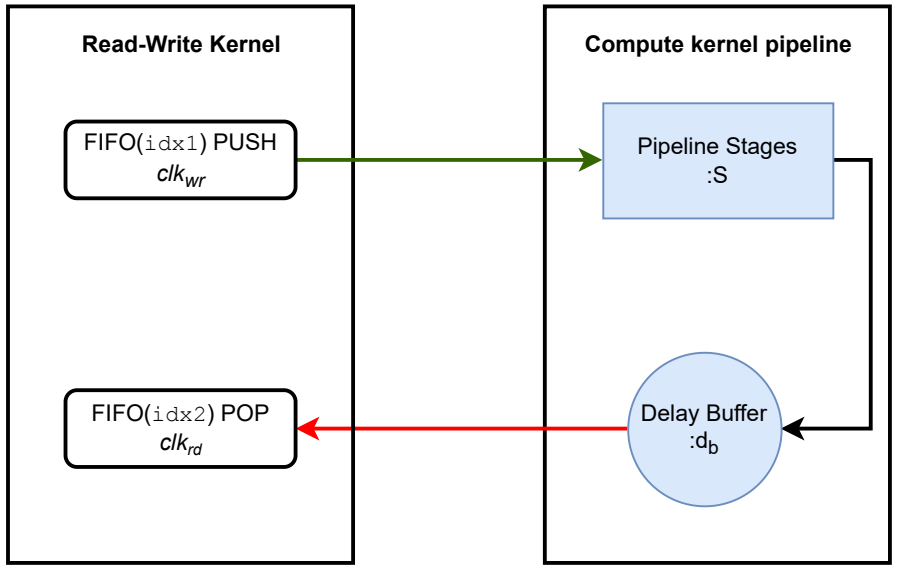
SYCL: Moving Iterative Loop to FPGA

- Deadlock due to runtime scheduling
  - Fuse read kernel and write kernel
- Delay pipe read (`idx2`) by number of iterations
  - Pipe operations are blocking
  - Avoids deadlock or throughput reduction due to waiting
- Value for delay depend on
  - Hardware schedule of pipe read and writes
  - Latency due to register stages and delay buffering

```
1  [[intel::disable_loop_pipelining]]
2  for(int itr = 0; itr < n_iter; itr++) {
3    accessor ptrR = ((itr & 1) == 0) ? in : out;
4    accessor ptrW = ((itr & 1) == 1) ? in : out;
5    [[intel::ivdep]] [[intel::initiation_interval(1)]]
6    for(int i = 0; i < total_itr+delay; i++) {
7      struct dPath16 vecR = ptrR[i+delay];
8      if(i < total_itr) pipeM::PipeAt<idx1>::write(vecR);
9      struct dPath16 vecW;
10     if(i >= delay) vecW = pipeM::PipeAt<idx2>::read();
11     ptrW[i] = vecW;
12   }
13 }
```



Delay Model



$$delay > clk_{wr} - clk_{rd} + S + d_b$$

Data will be available when FIFO pop is attempted

$$delay < clk_{wr} - clk_{rd} + S + d_b \text{ and } delay > clk_{wr} - clk_{rd} + d_b$$

data will be available after few clocks from first attempt of FIFO pop, Leading to reduced throughput

$$\text{delay} < \text{clk}_{\text{wr}} - \text{clk}_{\text{rd}} + \text{d}_{\text{b}}$$

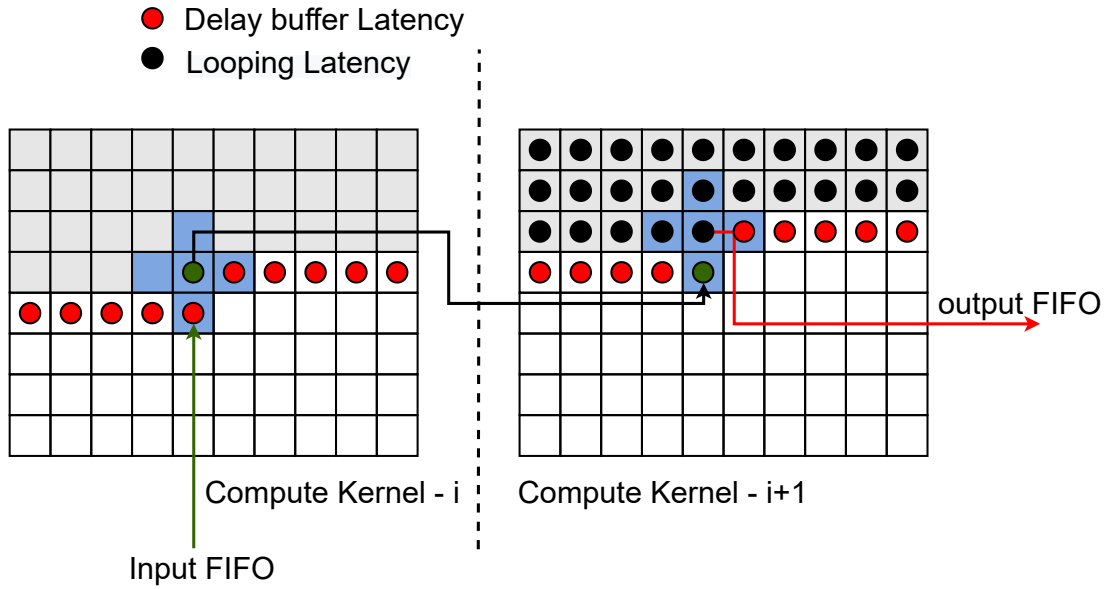
Data never going to be available to pop from the FIFO

Model: Performance

- Two main components of the latency
  - Computation latency in looping through mesh
  - Cascaded compute module delay buffer latency
  - Latency due to hardware pipeline

- 2D Mesh runtime model
  - $delay_{2d} = (S_{2d} + d_{b,2d})$
  - $S_{2d} = \sum_{i=0}^{\#\ kernels}(clk_{wr,i} - clk_{rd,i})$
  - $d_{b,2d} = \left\lceil \frac{m}{V} \right\rceil \times p \times \frac{D}{2}$
  - $Clks_{2D} = \frac{n_{iter}}{p} \times \left(\left\lceil \frac{m}{V} \right\rceil \times (n \times B + delay_{2d}) \right)$

- Similar model for 3D application is on the paper
- Over 85% accuracy



Symbol	Parameter
V	Vectorization Factor
p	Iterative loop unroll factor
D	Stencil Order
m, n, l	X, Y, Z dimensions of mesh
B	Batch Size
n_{iter}	Total number of iterations

Application Class 2 – Multidimensional Tridiagonal Solvers

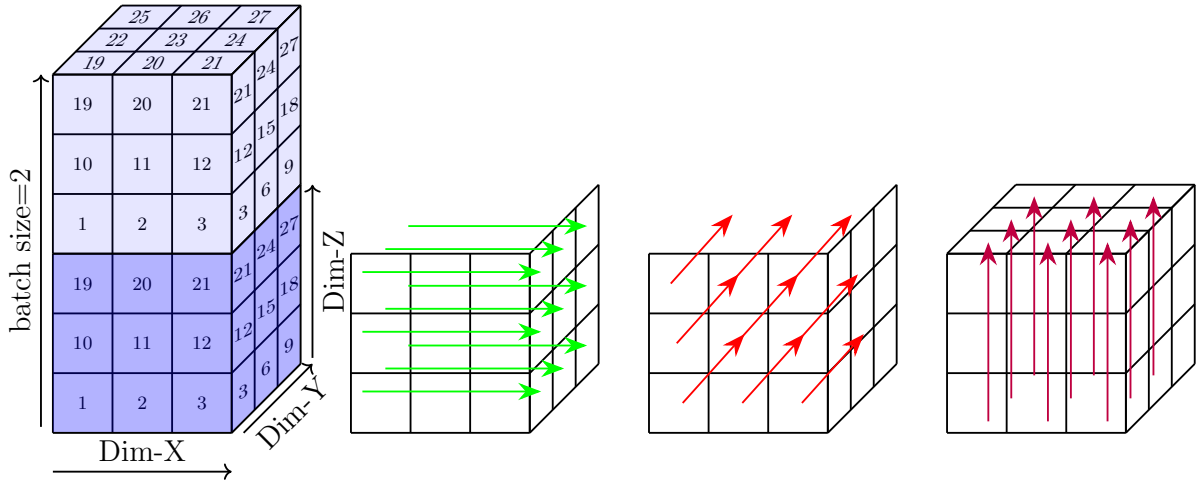
- Common in applications solving partial differential equations using implicit schemes
  - Computational fluid applications
  - Financial computing – option pricing

- Popular Tridiagonal System Solvers
  - Thomas algorithm
  - PCR
  - SPIKE

- Popular Alternating Direction Implicit(ADI) time method
  - Solves multiple systems along the coordinates
  - Work in Kamalakkannan et al. ICS2022 shows how the Thomas algorithm is the more efficient for multiple solves

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

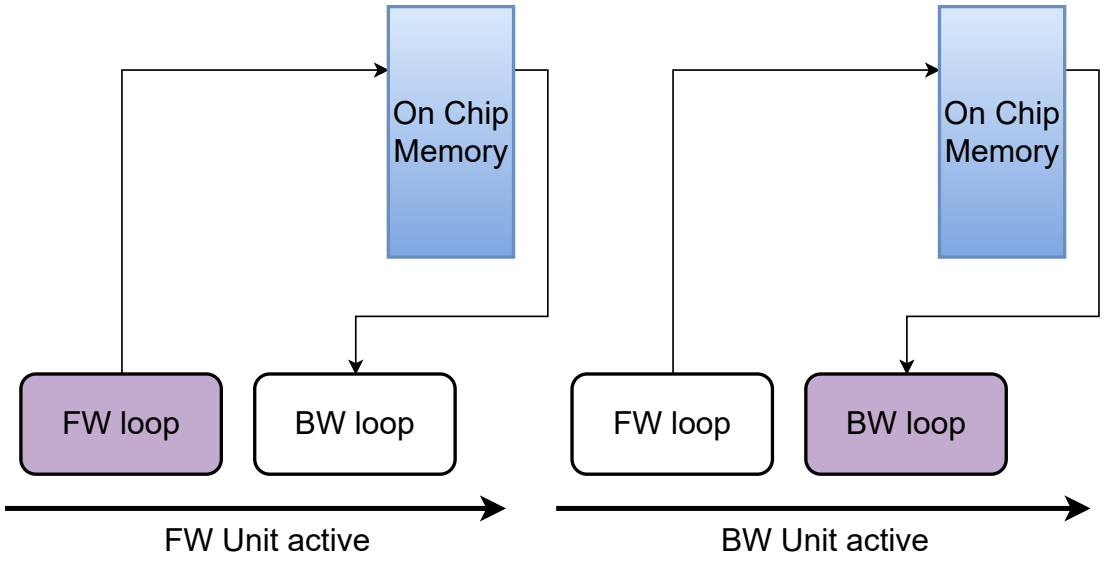
$$\begin{bmatrix} b_0 & c_0 & 0 & \dots & 0 \\ a_1 & b_1 & c_1 & \dots & 0 \\ 0 & a_2 & b_2 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{N-1} & b_{N-1} \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{N-1} \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\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



(a) Batched mesh (b) X-solve (c) Y-solve (d) Z-solve

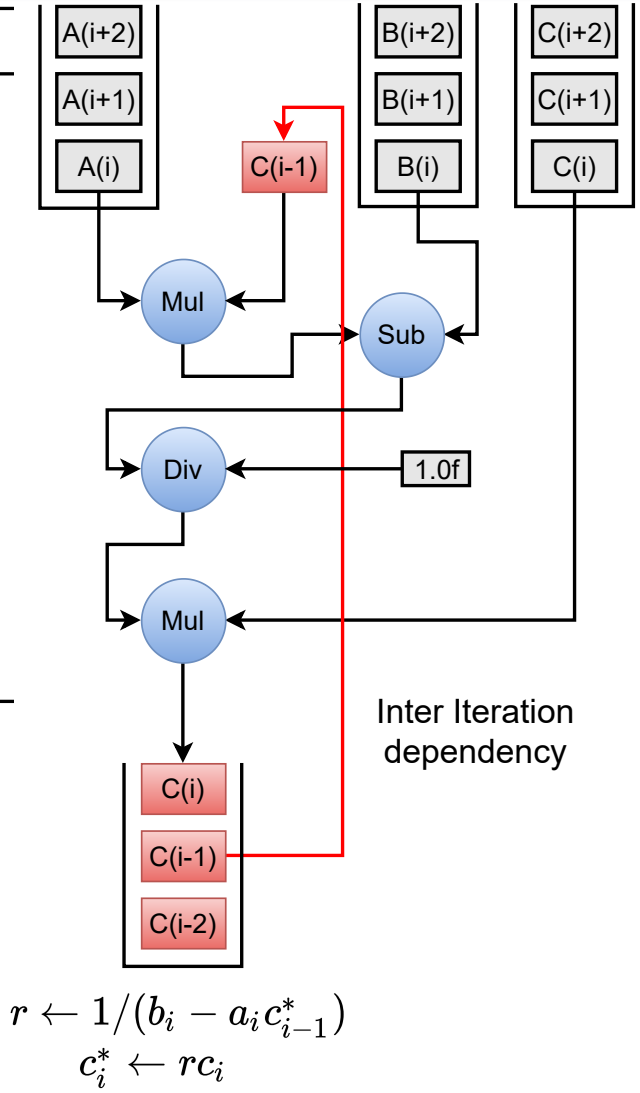
Implicit solvers: Thomas Solver

- Dependency
  - Between iterations in forward loop and backward loop
  - Between two loops
- Can't achieve initiation interval II=1
  - Floating point arithmetic operations are multi clock cycle
  - If iteration latency for FW loop $l_f$and BW loop l_b
  - Arithmetic pipeline is not effectively used
  - Total number of clock cycles is: $N \times (l_f + l_b)$
- FW loop and BW loop executes one after another



Algorithm 1: $\texttt{thomas}(a, b, c, d)$

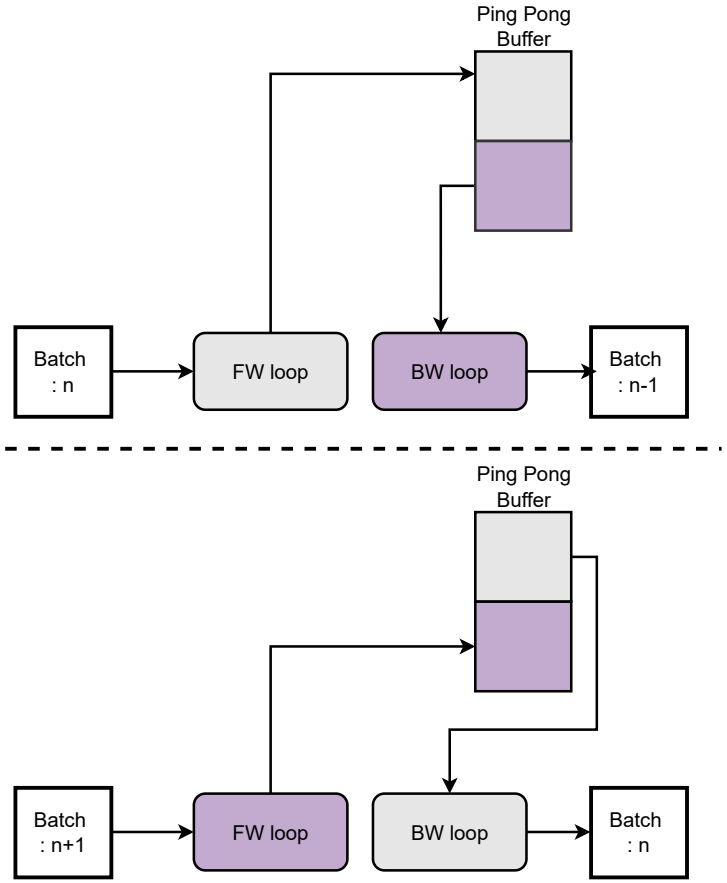
$d_0^* \leftarrow d_0 / b_0$
$c_0^* \leftarrow c_0 / b_0$
for $i = 1, 2, ..., N-1$ **do**
  $r \leftarrow 1/(b_i - a_i c_{i-1}^*)$
  $d_i^* \leftarrow r(d_i - a_i d_{i-1}^*)$
  $c_i^* \leftarrow r c_i$
end for
for $i = N-2, ..., 1, 0$ **do**
  $d_i \leftarrow d_i^* - c_i^* d_{i+1}$
end for
return d



$r \leftarrow 1/(b_i - a_i c_{i-1}^*)$
$c_i^* \leftarrow r c_i$

Thomas Solver: Batching and Dataflow Optimizations

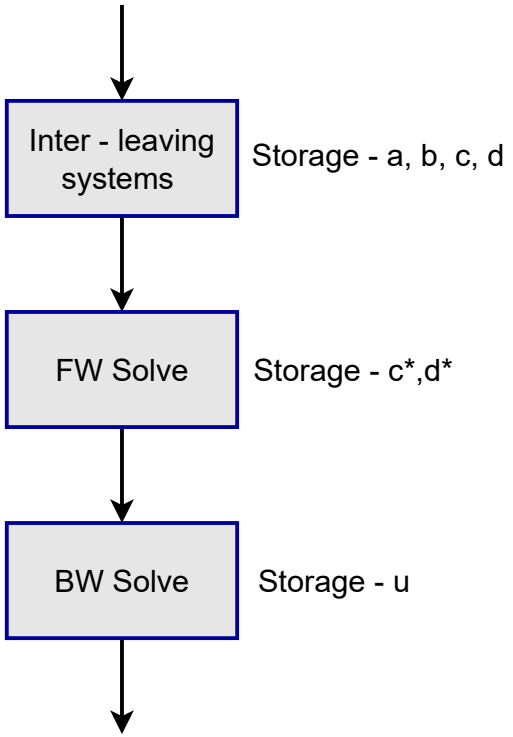
- Group of Systems can be solved interleaved manner
  - Fully utilizing arithmetic pipeline
  - group size should be equal or higher than iteration latency l_f / l_b
  - Assume group size $g = max(l_f, l_b)$
- Improves the average clocks to *N* for each loop
- Latency to solve single system will remain same
- Double buffering to execute FW loop and BW loop in parallel
  - Dual port memory
  - Separate partition for memory read and write



THOMAS SOLVER: ON-CHIP MEMORY REQUIREMENT

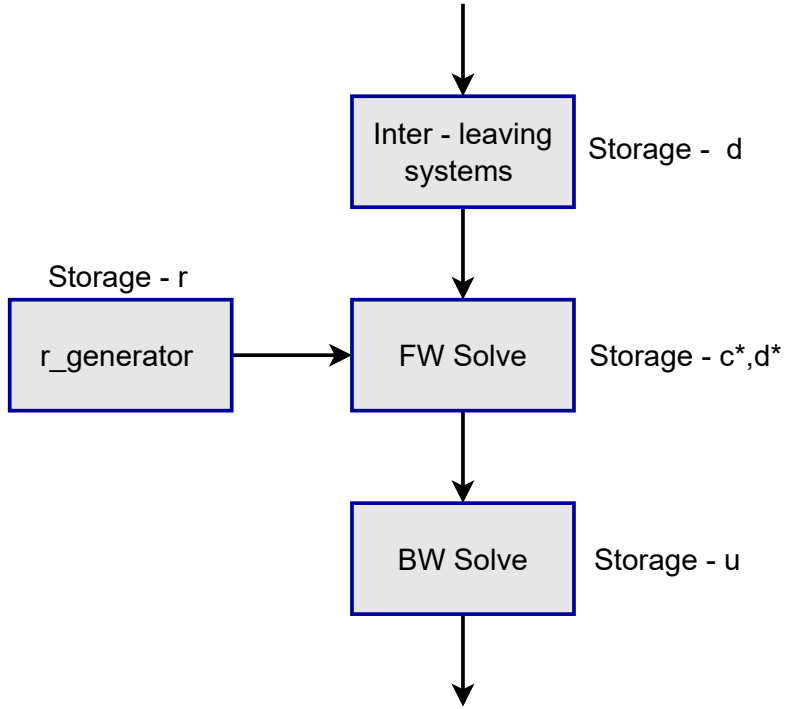
- On-chip memory is the key and limiting resource when solving reasonably larger system on FPGA
  - Each data structure requires *2gN* words
  - *g* number of interleaved systems with size *N*
  - Twice memory requirement due to ping pong buffer

$$memory\ cost = 7\ RAMs \times 2gN\ words/RAM$$



Thomas Solver: On-chip memory Optimization

- Coefficient (`a,b,c`) can be calculated for some implicit applications
  - Coefficient generation can be fused to Thomas forward solve
  - Saves storage required for (a,b,c) in Thomas interleave
  - 49% reduction of on chip memory requirement



- Reducing required group size will help to save on-chip memory.
  - Decoupling high latency floating-point based computation r to separate kernel
  - Having only DSP supported FP ADD/SUB/MUL in other three kernels reduce group size
  - Save 40% on chip memory compared to fused version.

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**

Performance – Intel PAC D5005 vs. Nvidia V100

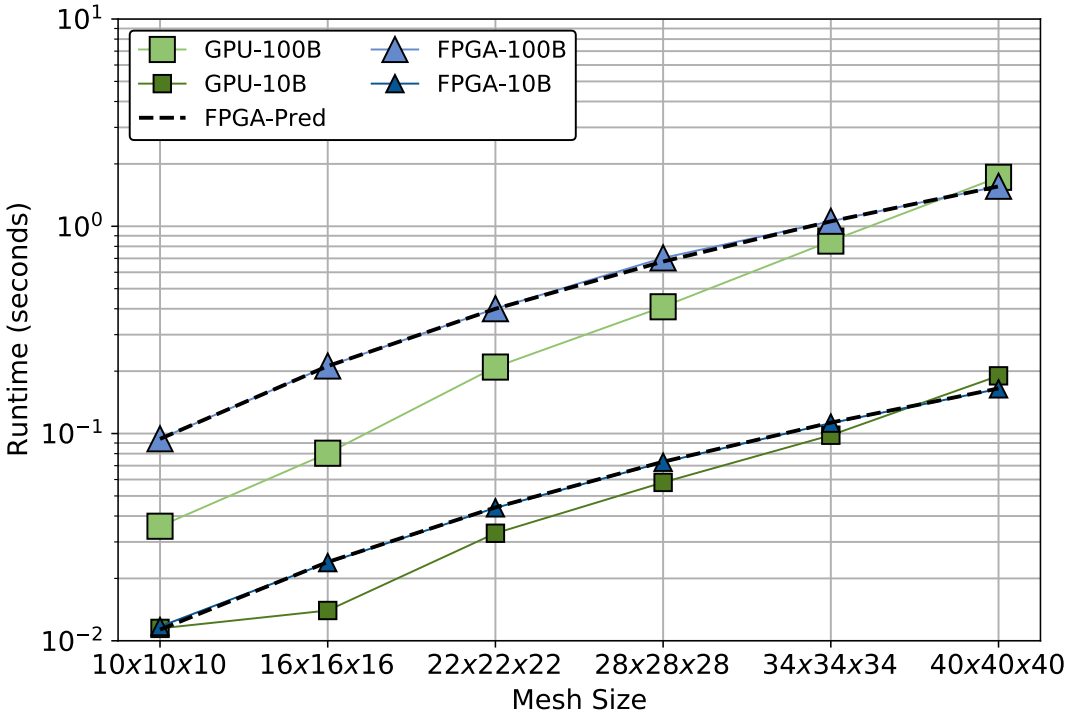
- Two representative applications
  - RTM_forward – 3D, 25-point stencils, vector elements
  - 2D ADI FP32 – 2D Heat Diffusion Equation using alternating direction implicit method
- FPGA kernels implemented using SYCL (DPC++)
- GPU implementation using CUDA
  - RTM forward using OPS framework
  - 2D ADI FP32 using Trid solver library https://github.com/OP-DSL/tridsolver
  - Equivalent or better performance than NVIDIA cuSPARSE
- Comparison – Intel PAC D5005 Vs Nvidia V100
  - Time to solution
  - Bandwidth
  - Power
- Fair comparison – GPU is saturated by batching

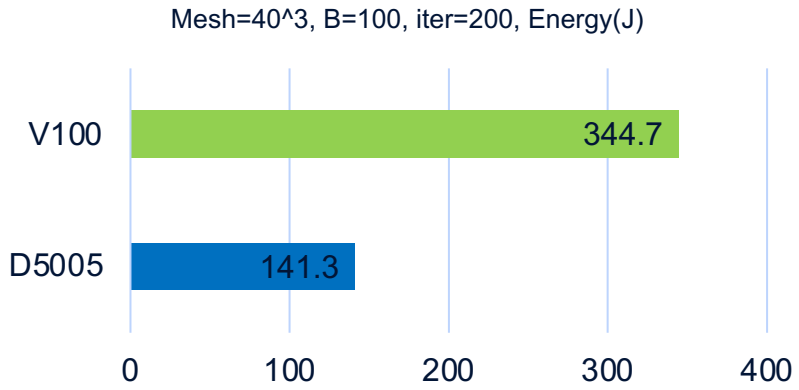
FPGA	Intel PAC D5005
DSP blocks	5760
MLABs / M20K	7.6MB / 29.3 MB
DDR4	64GB, 76.8GB/s, in 4 banks (1 channel/bank)
Host	Intel Xeon Platinum 8256 @3.8GHz (16 CPUs, 4 cores each) 1559 GB RAM, Ubuntu 18.04.6 LTS
Design SW	Intel oneAPI 2021.4.0, Intel Quartus software 19.2
board_variant	pac_s10
GPU	Nvidia Tesla V100 PCIe
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

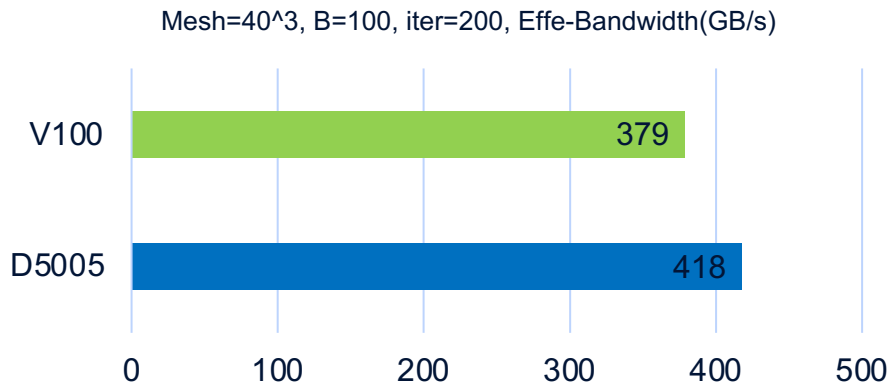
	D5005	V100
TeraFLOPS (FP32)	9.2	14
Memory Bandwidth (GB/s)	76.8	900
PCIe Bandwidth (GB/s)	32	32

RTM_forward Results – Intel D5005 Vs Nvidia V100

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**

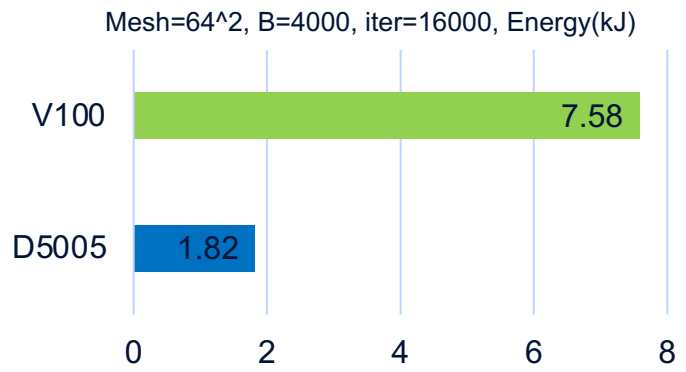


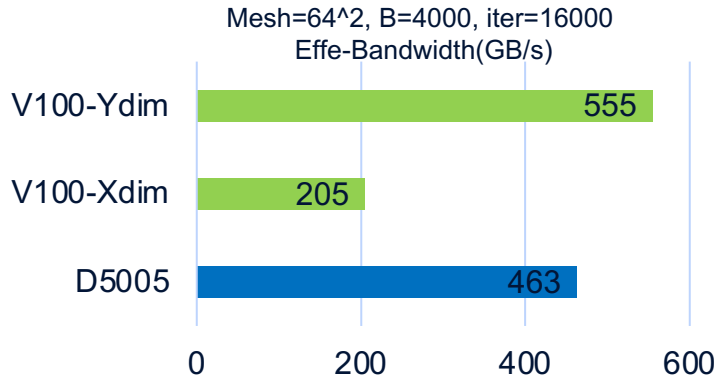


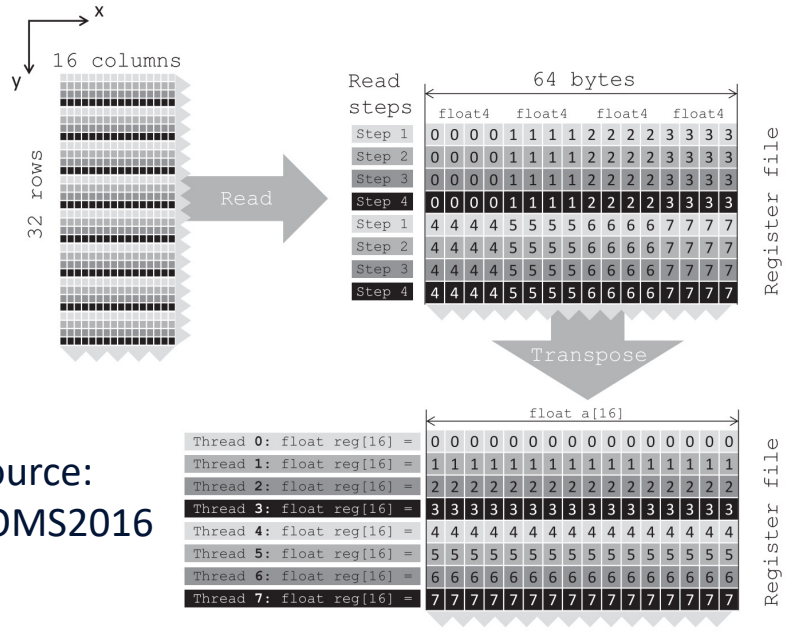


2D ADI FP32 – INTEL D5005 VS NVIDIA V100

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**

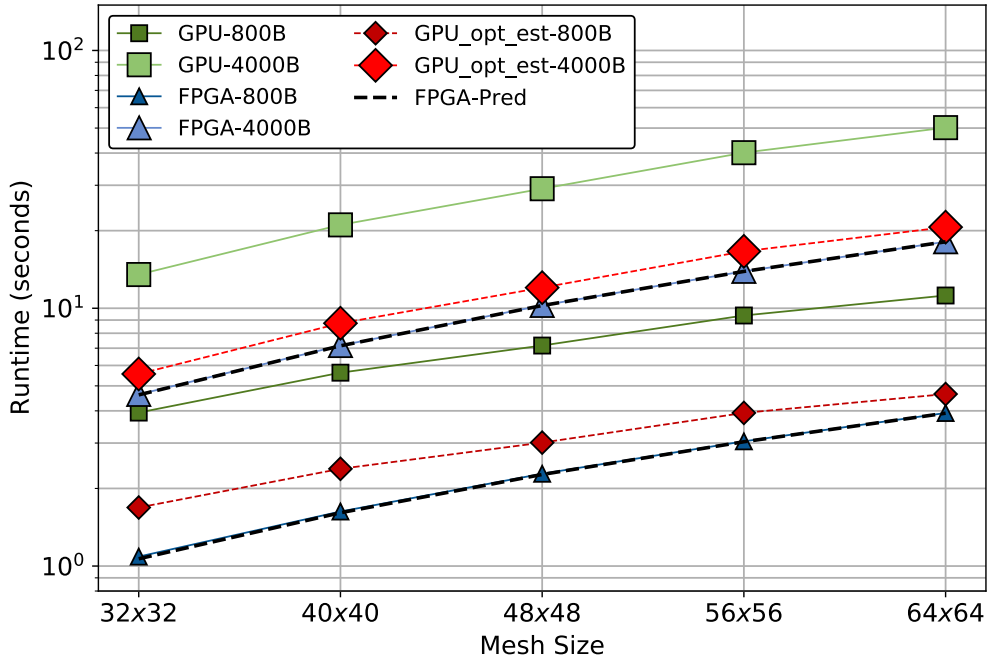






Source: TOMS2016

Fig. 9. Local transpose with registers.



Lessons Learnt

- SYCL offer Functional portability, but significant customization is required to gain decent performance on FPGAs
  - Kernel to Kernel communication to overcome global memory bandwidth
  - Custom memory hierarchy design for better data reuse
- Significant effort required for applying non-trivial transformation to get optimized SYCL implementation
  - Programming overheads still dominating on FPGAs
- FPGAs could challenge GPUs for (small) subset of HPC applications
  - Competitive or better performance on FPGA compared to current best traditional architecture (GPUs)
  - Significant energy saving on FPGA

Conclusion and Next Steps

- Used SYCL to design and develop structured mesh solvers on Intel FPGAs
  - Structured mesh based explicit stencil solvers – workflow for any stencil solver
  - Multiple multidimensional tridiagonal system solvers - best implementation based on Thomas algorithm
- Key Challenges for near optimal performance
  - Reducing kernel call overhead my moving iterative loop to FPGA
  - Reducing on chip memory usage by decoupling computation in Thomas forward loop
- Performance of solutions synthesized on an Intel D5005 FPGA
  - designs for non-trivial, production representative applications
  - Competitive performance compared to optimal implementation of same applications on Nvidia-V100
  - 59%-76% energy saving on FPGA for largest configuration of each application case
- Next steps
  - Automating code generation of structured mesh applications to target FPGAs using SYCL
  - Exploring the performance on Intel FPGAs with HBM

Stencil Solvers : https://github.com/Kamalavasan/StencilsOnFPGA
Tridiagonal Solvers : https://github.com/Kamalavasan/Tridsolver-FPGA

References and Acknowledgements

Acknowledgements

- Gihan Mudalige was supported by the Royal Society Industry Fellowship Scheme (INF/R1/1800 12)
- Istvan Reguly was supported by National Research, Development and Innovation Fund of Hungary(PD 124905)
- We are grateful to Intel for providing access to devcloud
- We are grateful to Jacques Du Toit and Tim Schmielau at NAG UK for the RTM application

References

- Waidyasooriya2017: H. M. Waidyasooriya, Y. Takei, S. Tatsumi and M. Hariyama, "OpenCL-Based FPGA-Platform for Stencil Computation and Its Optimization Methodology," in IEEE Transactions on Parallel and Distributed Systems, vol. 28, no. 5, pp. 1390-1402, 1 May 2017, doi: 10.1109/TPDS.2016.2614981.
- Soda2018: Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: stencil with optimized dataflow architecture. In Proceedings of the International Conference on Computer-Aided Design (ICCAD '18). Association for Computing Machinery, New York, NY, USA, Article 116, 1–8. DOI:https://doi.org/10.1145/3240765.3240850
- 3dRTM2011: Haohuan Fu and Robert G. Clapp. 2011. Eliminating the memory bottleneck: an FPGA-based solution for 3d reverse time migration. In Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA '11). Association for Computing Machinery, New York, NY, USA, 65–74. DOI:https://doi.org/10.1145/1950413.1950429
- IPDPS2021: Kamalakkannan, K., Mudalige, G.R., Reguly, I.Z. and Fahmy, S.A., 2021, May. High-level FPGA accelerator design for structured-mesh-based explicit numerical solvers. In 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS) (pp. 1087-1096). IEEE.
- ICS2022: Kamalakkannan, K., Reguly, I. Z., Fahmy, S. A., and Mudalige, G. R. (2022). High Throughput Multidimensional Tridiagonal Systems Solvers on FPGAs. In Proceedings of the ACM International Conference on Supercomputing (ICS ‘22). Association for Computing Machinery, New York, NY, USA (Accepted)
- TOMS2016: Endre László, Mike Giles, and Jeremy Appleyard. 2016. Manycore Algorithms for Batch Scalar and Block Tridiagonal Solvers. ACM Trans. Math. Softw. 42, 4, Article 31 (July 2016), 36 pages. https://doi.org/10.1145/2830568