Adding point-to-point communication between FPGAs to an accelerator for the Discontinuous Galerkin method

MASTERARBEIT

Submitted towards partial fullfilment
of the requirements for the degree of

Master of Science

at University of Paderborn

by

Gaurav Kumar Singh

Advisor:

Dr Tobias Kenter,
Prof. Dr. Jens Förstner

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Faculty for Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group High Performance IT Systems

Mai 2019

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Paderborn, May 8, 2019

Gaurav Kumar Singh

Contents

Abstract

This should be a 1-page (maximum) summary of your work in English.

Chapter 1

Introduction

Computational efficiency of algorithms have reached almost a peak over the years and further improvements using system level parallelism such as multiprocessing, multi-core systems or instruction level parallelism such as pipelining etc. has also not proven to significantly contribute towards further improvement. These limitations have led to use of parallel multi-core processing systems which use multiple CPUs, GPUs and FPGAs to speed up the processing of applications which require high computational performance. These systems, generally referred as supercomputers or high performance clusters, comprises of multiple individual systems known as nodes. Each node can contain multiple multi-core processing, GPUs and FPGAs and are connected to each other using high speed Ethernet to exchange data and control information. The whole system acts as a single high performance system with distributed memory and processing elements as shown in fig

add image

Traditionally multiple CPU based supercomputers were only utilized but considering the nature of the application which mostly consists of the same operation performed on a stream of data, GPUs which could do multiple of these operations in parallel Single Instruction Multiple Data (SIMD) were utilized to increase performance further. The GPUs act as accelerators and are attached to main CPUs which offloads vector based computation suitable for GPU via special software constructs. The GPUs help in reducing a lot of computation time for large amount of data.Though the CPU and GPU based parallel architecture provide significant performance benefits, these are limited by the size of the cluster and the problem size. These limitations led researchers to look for technologies which would help in decreasing the execution time for single step even further by using hardware based accelerators.

FPGA based accelerators are thus seen as the most prominent alternative. FPGAs provide the flexibility to design application specific hardware accelerators and re-use the same FPGA for different kinds of problem without huge programming overheads. FPGAs as an accelerator is mostly used to implement simple mathematical operations such as matrix multiplication, fast fourier transformation and string manipulation which form the basis for most of the application. Using the FPGA's features such as data and instruction pipelining, replication and memory localization, the FPGAs can be used to decrease the execution time of these mathematical operations.

Increasing popularity of the FPGAs in the high performance computing has led to new innovations for the FPGAs. Higher memory bandwidths, increased count of logical units and bigger local memory are some of these. The new Noctua high performance computing (HPC) cluster at Paderborn Center for Parallel Computing (PC^2) contains the current generation of Intel Stratrix 10 FPGAs which provide specialized high speed serial IO channels which can be

used to set up communication infrastructure between the FPGAs. The Nallatech 520N boards used are equipped with four 100/40/25/10G QSFP28 network ports which use these serial IO channels and would provide a very high speed communication setup and opportunities to scale up application over multiple FPGAs with small communication latency replacing applications which use MPI for communication.

MIDG2[1] is an MPI based parallel computation implementation of the Discontinuous Galerkin (DG) [11] method to solve Maxwell's equation in time domain. DG method is a commonly used operation in many simulation applications to solve partial differential equations (PDE) and improvements to the computation time would be an important step to benefit different simulation applications utilizing this method. This thesis presents a design which evaluates the benefits of using a parallel distributed FPGA based system to reduce the execution time of the application by using direct FPGA-to-FPGA communication.

1.1 Objectives

The Noctua being first of the academic HPC cluster equipped with FPGAs provides opportunities to create systems which are accelerated with multiple FPGAs. As there is no known implementation utilizing FPGA in network configuration for accelerating the DG method, this master thesis aims at presenting such a system to evaluate the achievable acceleration with multiple FPGAs using point-to-point communication in different network topologies for the MIDG2 application. Towards achieving this target, the evaluation can be divided into two stages. First stage aims at identifying and evaluating the possible topologies for point-to-point communication. In the second stage of the thesis, the existing openCL™ based FPGA implementation for the MIDG2 application would be extended to use IO channels.

Considering the two stages as individual sub-tasks, the first task involves building prototypes using openCL™ IO channels which would utilize the four 100/40/25/10G QSFP28 network ports available on the Nallatech 520N boards. These prototypes would be used to evaluate the point-to-point topologies using bandwidths and latency to give an overview of possible benefits. The prototypes would also serve as the basis for understanding the implementation opportunities available in the openCL™ and Nallatech BSP for the channels. The second task would be then to utilize the understanding and extend the openCL™ kernels for the MIDG2 application such that the need for MPI communication to communicate information or the shared surfaces is eliminated and overheads involved the removed.

1.2 Related Work

DG method is believed to be first proposed by Reed and Hill [18] for solving the steady-state neutron transport equation [11]. Over the years many researchers have proven the accuracy of the method and developed improvements over the original method to use in different fields for solving PDE. The DG method is popular today to solve equations in the fields of acoustics [1, 22, 23], elasticity [9, 12, 13], electrodynamics Maxwell's equation [4–7, 17] and thermodynamics [8].

The popularity of the DG method has led researchers to develop parallel methods which can speed up the application to reduce the execution time. As the DG method performs operations on local elements followed by accumulation, this allows parallelization of the operations on elements and has been utilized to create various parallel systems. Baggag, Atkins, and Keyes [2] presents a parallel system which is used to simulate aeroacoustic scattering and uses MPI data

[1] https://github.com/tcew/MIDG2

exchange. Klöckner et al. [15] showed the benefits of using GPU for accelerating the computation by utilizing the capability of the GPU to process data in parallel. They use CUDA programming for GPUs to get improved memory bandwidths and higher computation efficiency. Bernacki et al. [3] discuss the benefits of parallelization achieved by partitioning the tetrahedral meshes for realistic problems involving electromagnetic wave radiation study for different objects. Though the use of such parallel architecture and use of GPU based accelerators have proven to improve the simulation time, they often require large data sets to show the benefits of such improvements.

FPGAs based accelerators allows to overcome such issue as they provide architecture such as efficient arithmetic operations using DSPs, deep instruction and pipelining to decrease computation time, replication for large scale parallel computations. Such benefits have made FPGAs a popular choice for implementing accelerators for problems utilizing heavy floating point computation. Considering such benefits, an implementation for accelerating the DG operations in FPGAs was developed by Kenter et al. [14]. The implementation shows the benefit of using a single FPGA over a highly parallelized multi-core CPU implementation for the MIDG2 application which solves Maxwell's equations in time domain. An extension of this design is implemented by PC2 which is capable or using multiple FPGAs using MPI. This design is used as the base for this thesis.

The increased popularity of the FPGAs in past as led to identify possibilities for using multiple FPGAs. This can be achieved by using MPI communication via the host to which the FPGA is connected. Though such design posses benefits in many cases, it performs poorly in case of multiple and large transfer due to very low bandwidths for the PCIe + MPI combination. Systems have been investigated where the FPGAs could communicate directly without the need of host. Sheng et al. [20] presents a design for 3D FFT solver which uses a 3D-torus FPGA-based network using a table-based routing scheme. The evaluation results of such design shows 30% improvement over the reference design. Kobayashi et al. [16] used an openCL™ based design implementation to compare the latency of MPI+PCIe based system and FPGA-to-FPGA system which uses Ethernet IP for communication over an switch. The results show a large difference between the achievable bandwidths and latency for the two system proving the benefits of such system. The system design presented in this thesis uses a similar approach like in [16] but does not utilize the Ethernet-IP core and relies on the serial communication support provided in the Nallatech BSP.

Update this description if the chapter structure changes

The rest of the thesis is divided into 4 chapter. Chapter 2 would discuss the fundamentals of the components and technologies used in the thesis. Chapter 4 would present the system design and details of the implementation done as part of the thesis for both the tasks giving reasons for various approaches and discussing the evaluated systems or designs. Chapter 6 discusses the evaluation steps and the results of the evaluation in detail highlighting the benefits of using the IO channels. Final chapter 7 presents a conclusion of the achieved results of the thesis and proposes possible future works and improvements.

Chapter 2

Fundamentals and Related Work

This chapter would discuss the details of the various components involved highlighting the use of the components for the completion of this thesis. The last section would give a overview of related work which provide some details of systems which are either partially related to this thesis or form the basis for the thesis to improve upon.

2.1 Nodal Discontinuous Galerkin Method

The Nodal Discontinuous Galerkin Method in Time Domain (DGTD) [11] is used to find solutions for partial differential equations (PDE) numerically. This method is efficient in producing results with computers as it relies on mathematical calculations on elemental basis. This allows to perform computation in parallel on similar or different hardware helping to solve problems from different domains quickly. DGTD method is particularly popular for applications in the domains such as fluid mechanics, plasma physics and electrodynamics.

2.1.1 DGTD to solve Maxwell equations

Hesthaven and Warburton [10] presented the use of DGTD to solve time-domain Maxwell's equations with an 1D and 2D examples and the extension to 3D is explained in [11]. This section would briefly describe the formulations and steps involved for getting the solutions which are used as the basis for implementation in the application used in this thesis to simulate electromagnetic flux values for a given material.

The basic equation involved in computation of the electromagnetic flux is Maxwell's equations. The three dimensional time-dependent Maxwell's equations [11] is written as:

$$\mu\frac{\partial \mathbf{H}}{\partial t} = -\nabla \times \mathbf{E}, \varepsilon\frac{\partial \mathbf{E}}{\partial t} = -\nabla \times \mathbf{H} \tag{2.1}$$

the conservation form of the equation can be expressed as:

$$\mathcal{Q}\frac{\partial \mathbf{q}}{\partial t} + \nabla \cdot \mathcal{F} = 0 \tag{2.2}$$

where

$$\mathbf{q} = \begin{bmatrix}\mathbf{H}\\ \mathbf{E}\end{bmatrix}, \mathcal{Q} = \begin{bmatrix}\mu & 0\\ 0 & \varepsilon\end{bmatrix}, \mathcal{F} = \begin{bmatrix}-\widehat{n} \times \mathbf{E}\\ \widehat{n} \times \mathbf{H}\end{bmatrix} = \begin{bmatrix}\mathbf{F}_H\\ \mathbf{F}_E\end{bmatrix} \tag{2.3}$$

$\mathbf{H}$ and $\mathbf{E}$ are the magnetic and electric vector fields in three dimensions which are function of the positional coordinates and time $(\tilde{x}, \tilde{y}, \tilde{z}, \tilde{t})$, $\mathcal{Q}$ defines the magnetic permeability $\mu(x)$ and the electric permittivity $\varepsilon(x)$ of the material.

Now considering that we have a object composed of a single type of material with known $\mathcal{Q}$ values, the DG method can be used to solve the equation 2.2 by discretization of the computation domain Ω (whole of the object) spatially. In 3D, this can be achieved by dividing the object into K tetrahedral elements and computing the local approximated solution $u_h^k(x,t)$ for each element $D^k \in K$. This local solution is computed for a defined polynomial order N such that, $h \in N$ represents the h^{th} polynomial of D^k and is called the nodal point. Now for a given polynomial order N, the local solution can be expressed [11] as:

$$x \in D^k \ : \ u_h^k(\mathbf{x},t) \ = \ \sum_{n=1}^{N_p} \hat{u}_n(t)\psi_n(\mathbf{x}) \ = \ \sum_{i=1}^{N_p} u_h^k(\mathbf{x}_i,t) l_i^k(\mathbf{x}) \tag{2.4}$$

where $l_i^k(\mathbf{x})$ is the multidimensional Lagrange polynomial and $\psi_n(\mathbf{x})$ is a three-dimensional polynomial basis.

In equation 2.4, the N_p denotes the number of nodal points per element D^k and depends on the polynomial order N which is given by:

$$N_p = \frac{(N+1)(N+2)(N+3)}{6} \tag{2.5}$$

Now that we know the basis for computation of local approximated solutions, the next step is to compute the global solution, which can be achieved by summing up the these individual solutions $u_h^k(x,t)$ except for nodal points on the faces. The nodal points on the faces of the tetrahedra would have two different solution as they can be part of two different D^k elements and required coupling of the solution. The coupling of the solution is done by computing the electric and magnetic field differences [14] $\Delta\mathbf{E} = \mathbf{E}^+ - \mathbf{E}^-, \Delta\mathbf{H} = \mathbf{H}^+ - \mathbf{H}^-$. Combining the local and the global solutions is achieved [4, 14] by a pair of ordinary differential equations (ODE) for the semi-discrete system derivation or which is explained in [11]:

$$\epsilon^k \frac{\partial \mathbf{E}^k}{\partial t} = \mathbf{D}^k \times \mathbf{H}^k + (\mathcal{M}^k)^{-1}\mathcal{F}^k \left(\frac{\Delta\mathbf{E} - \hat{n}\cdot(\hat{n}\cdot\Delta\mathbf{E}) + Z^+\hat{n}\times\Delta\mathbf{H}}{\overline{Z}} \right) \tag{2.6}$$

$$\mu^k \frac{\partial \mathbf{H}^k}{\partial t} = -\mathbf{D}^k \times \mathbf{E}^k + (\mathcal{M}^k)^{-1}\mathcal{F}^k \left(\frac{\Delta\mathbf{H} - \hat{n}\cdot(\hat{n}\cdot\Delta\mathbf{H}) - Y^+\hat{n}\times\Delta\mathbf{E}}{\overline{Y}} \right) \tag{2.7}$$

where $(\mathcal{M}^k)$ is mass matrix, $\mathcal{F}^k$ is face matrix, $\hat{n}$ outwardly pointing normal vector to the element face where the flux is calculated. The $Z^\pm$ and $Y^\pm$ is the impedance and the conductance of the material.

The solution of these ODEs require time discretization. The Runge-Kutta scheme introduced in [21] is used [11, 14] to integrate the equations in time. The timesteps are chosen in way such that they are small and ensure that the timestep error can be neglected. In the implementation used in this thesis, the timestep is computed by calculating the smallest distance between two nodal points.

2.2 Mini Discontinuous Galerkin Maxwells Time-domain Solver (MIDG2)

MIDG2 is a open source C/C++ based application which implements the DGTD method for solving Maxwell's equations for 1D, 2D and 3D. It uses K non-overlapping tetrahedra elements as described in section 2.1.1 for computing the flux values for a given object. This section will give an overview of the application implementation along with improvements done to use multiple FPGAs to offload the computation and speed up the execution time.

The original MIDG2 implementation supports parallelization using Message passing interface (MPI) for multiple CPUs and uses OCCA [1] to provide support for acceleration with GPU using CUDA and OpenCL™. The object for which the flux is to be computed is represented as an unstructured mesh of K non-overlapping tetrahedral elements as shown in fig 2.1. This mesh is generated using the tool Tetgen[2] which is an open source tool to generate meshes. The tetrahedra within the mesh are identified with their vertices. The application uses three vectors VX, VY and VZ to store the coordinates of each of the vertices. The application performs the steps which are formulated in section 2.1.1 using mesh and additional inputs which include mass matrixes and Runge-kutta time step constants to compute the flux values for a given polynomial order.

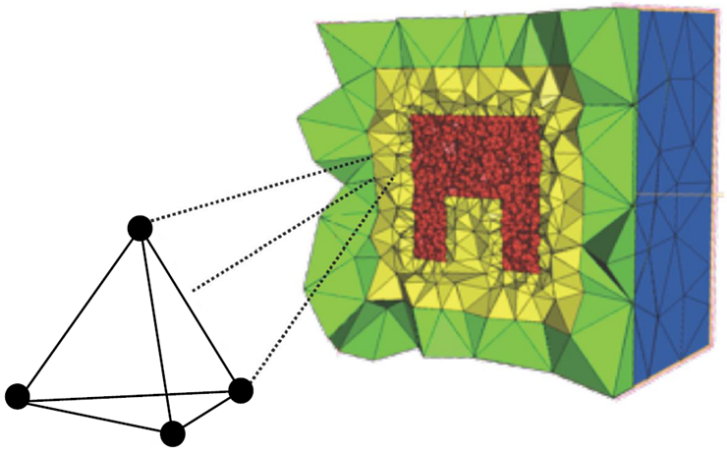


Figure 2.1: K element mesh with tetrahedra elements for split-ring resonator object

The parallelization or computation can be achieved in this implementation by dividing the mesh into partitions. As the DGTD algorithm works by first computing individual local solution for the elements and then accumulating the values over all $k's$, a partitioning scheme which uses surfaces as boundaries can be performed such that, computation of each individual partition is performed by a separate process/thread/core/system and the shared surface data is shared at each time step between them using a communication infrastructure. The partitioning in MIDG2 is achieved using an open source tool ParMETIS[3], which is effective in partitioning meshes for a distributed system for equal load sharing. The MIDG2 uses MPI for implementing a distribution and communication scheme which allows to use multiple systems with distributed load to speedup the computation. The whole process for partitioning and distribution is shown in fig 2.2.

2.2.1 FPGA implementation for DGTD

Kenter et al. [14] extended the original MIDG2 implementation to use FPGAs to accelerate the computation for a single FPGA system. This implementation uses Intel™ FPGA SDK for OpenCL™ to implement three compute kernels viz. volume kernel, surface kernel and RK kernel which perform the computation steps for the DG solver. The kernels developed were optimized to utilize the capabilities of the FPGA such as parallelization of operations by replication, optimizing memory access by using local memory for storing constant data, using memory banks to allow parallel data reads/writes for higher bandwidth lower stalls. The structure of

[1] https://libocca.org
[2] http://wias-berlin.de/software/tetgen/
[3] http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview

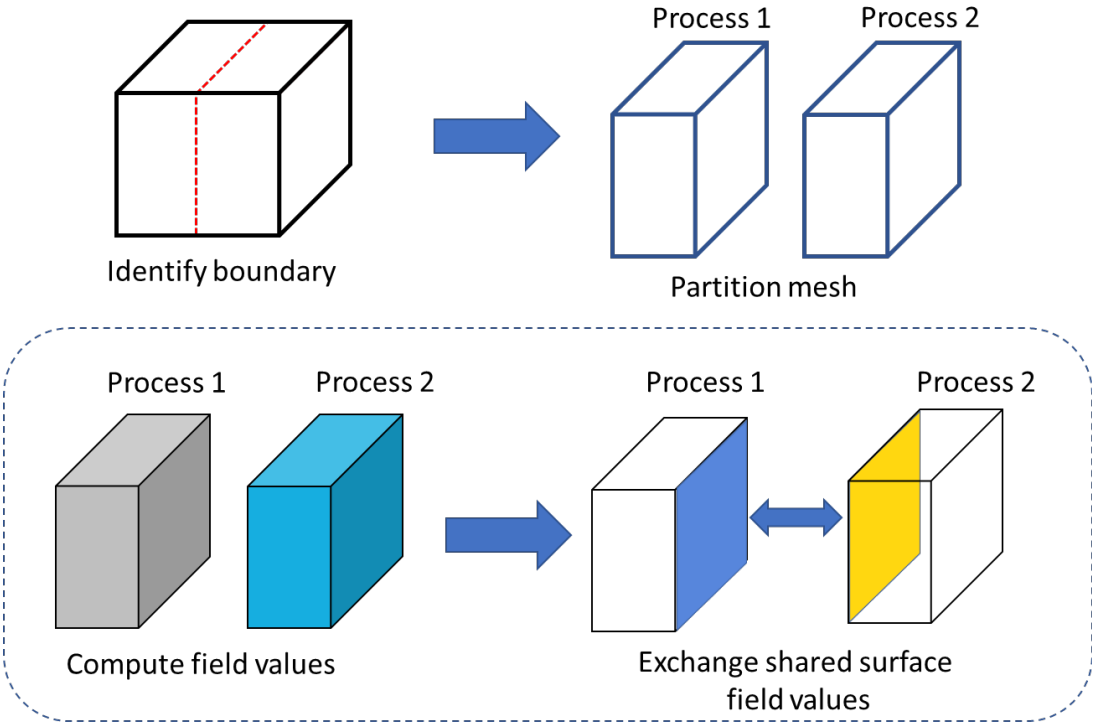


Figure 2.2: Parallelization achieved with partitioning and processing by multiple processes

the OpenCL™ kernels for this implementation is shown in figure. The IO kernels for surface and volume act as memory interface kernels which access the global memory as source and sink for data. Use for multiple kernels in such way along with the Intel™ FPGA OpenCL™ channels, create a pipeline structure for the volume and surface compute kernels which allows to start computation on a nodal point in each cycle. The kernel structure of the implementation is shown in figure 2.3 which shows pipeline structure developed for acceleration with the single FPGA. The complete implementation details along with performance evaluation for the design is described in [14].

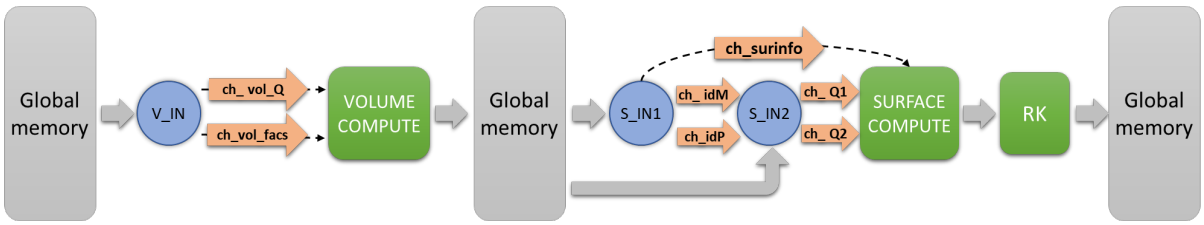


Figure 2.3: Block level structure for OpenCL™ kernels developed for single FPGA by Kenter et al. [14]

Further improvements to this design was made by extending the capabilities to work with multiple FPGAs along with MPI. The MPI FPGA design works similar to the original MIDG2 MPI implementation. The mesh is divided into partitions using ParMETIS and the computation is performed on different compute nodes of a cluster. The nodes in this case also include the FPGAs which is used to offload the computing and use the kernels from the FPGA based design. An additional kernel is introduced in this design which is responsible to read the shared data

and store it in a coalesced memory. The host processor then reads this data from the FPGA memory and uses MPI to communicate with other nodes to which the data is shared. Figure 2.4 shows the system architecture of this design.

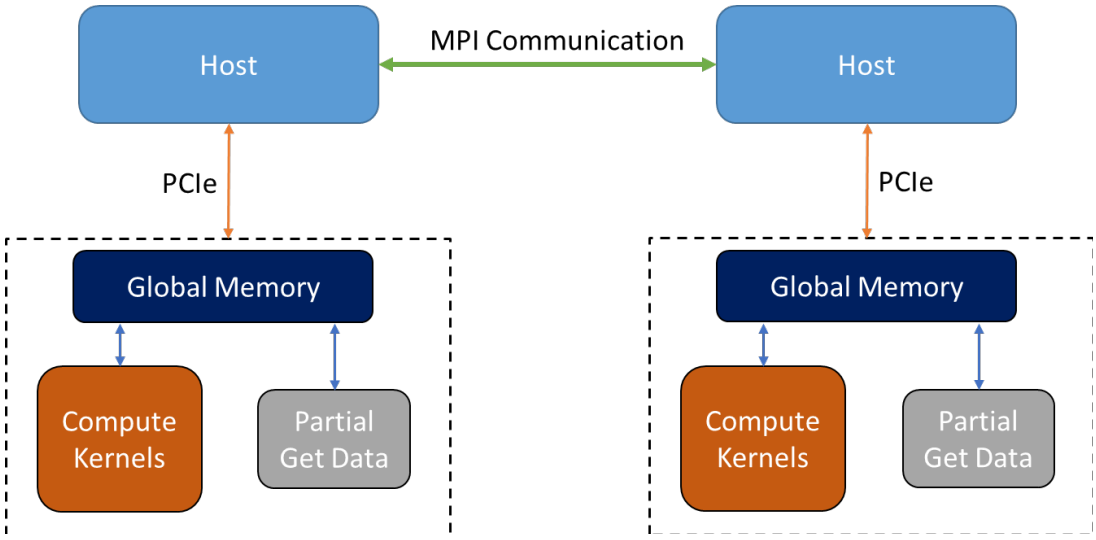


Figure 2.4: System level architecture for MPI FPGA design communicating using MPI and PCIe

Using this design multiple FPGAs can be used to speed up the computing even further. Though speed ups can be achieved with this design, the communication which involves movement of data through peripheral component interconnect express (PCIe) bus and the network interface two times, reduces the communication bandwidth a lot. This communication setup also requires to have synchronization constructs between the FPGA and the host processor to ensure data correctness which adds a lot of overhead in the overall execution time. This thesis uses this design as the base versions and replaces the communication with point-to-point communication between FPGAs which allows removing the complex communication and synchronization steps and speed up the execution even further.

2.3 Hardware and Software Platform

This section would describe the hardware and the software platform used in the thesis for implementing and evaluating the extended design. The first subsection gives an overview of Nallatech 520N board followed by and description of software setup using Intel™ FPGA SDK for OpenCL™ used to implement the kernels for the features.

2.3.1 Nallatech 520N

The Nallatech 520N boards used for the evaluation are equipped with

1. Intel™ Stratix 10 FPGA GX2800 capable for providing up to 10 TFLOPS of single precision floating point performance
2. DDR4 SDRAM memory divided into 4 banks with 8GB each giving a total memory of 32GB with a transfer rate of 2400 MT/s
3. Four 100/40/25/10G QSFP28 Network Ports
4. 16-lane PCI-Express Gen 3.0 for high speed host to FPGA data transfers

The Intel™ Stratix 10 FPGA GX2800 FPGA belongs to the current generation of the high-performance FPGA family with new Intel™ Hyperflex™ core architecture which gives higher bandwidth and processing performance to the FPGAs. The resource summary for the FPGA is given in table which shows the high computation capabilities of the FPGAs add table

QSFP Network Ports

Along with huge processing power, the Nallatech board also contains 4 QSFP Network Ports which are connected to the FPGA transceivers to provide high speed network communication. As mentioned previously these ports are used in this thesis to setup FPGA-to-FPGA networks to transfers data and reduce communication latency. The connections in the used setup are done using Finisar's FTL410QE2C and Edge's FTL410QE3C QSFP+ optical transceivers modules which support a maximum link length of 150 meters. The maximum data rate varies from 39.8 Gbits/s to 44.8 Gbits/s. As the Nallatech BSP supports 40 Gbits/s speeds, this is used for all calculation in this thesis.

2.3.2 Software Development flow

Intel™ FPGA SDK for OpenCL™ is used for the extending the existing kernels.

GIve details after finalizing

2.3.3 OpenCL™ Serial IO channels

The communication over the QSFP ports can be implemented in the OpenCL™ by using the Intel's OpenCL™ IO channels support. The IO channels can be used to stream data directly between kernels and I/O using explicitly named channels. The declaration of these channels should be included in the `board_spec.xml` using the `channels` element. The Nallatech BSP provide 4 Tx and 4 Rx IO channels for the 520N board as shown in the listing which interface to the 4 QSFP ports on the board. In order to use these I/O channels in the OpenCL kernel, an `io` attribute is included in the channel declaration along with id of the interface which is specified in the `board_spec.xml`.

Revise the introduction of the IO channels to describe that is it provided by Nallatech

```
<channels>
    <interface name="board" port="io_to_dev_ch0" type="streamsource" width="256" chan_id="
    kernel_input_ch0"/>
    <interface name="board" port="dev_to_io_ch0" type="streamsink" width="256" chan_id="
    kernel_output_ch0"/>
    <interface name="board" port="io_to_dev_ch1" type="streamsource" width="256" chan_id="
    kernel_input_ch1"/>
    <interface name="board" port="dev_to_io_ch1" type="streamsink" width="256" chan_id="
    kernel_output_ch1"/>
    <interface name="board" port="io_to_dev_ch2" type="streamsource" width="256" chan_id="
    kernel_input_ch2"/>
    <interface name="board" port="dev_to_io_ch2" type="streamsink" width="256" chan_id="
    kernel_output_ch2"/>
    <interface name="board" port="io_to_dev_ch3" type="streamsource" width="256" chan_id="
    kernel_input_ch3"/>
    <interface name="board" port="dev_to_io_ch3" type="streamsink" width="256" chan_id="
    kernel_output_ch3"/>
</channels>
```

A example kernel code is shown in the listing which uses one TX and one RX channel to communicate in the `sender` and `collector`. `ch_eth_in` and `ch_eth_out` is declared to interface with the external channels `kernel_input_ch0` and `kernel_output_ch0` respectively. The channels should be declared with a datatype (`float8` in this case) suitable to hold the 256 bits wide data which is the current width of the channels. After the declaration, the channels can be used by the kernel similar to standard OpenCL™ channels using `write_channel_intel` and `read_channel_intel` APIs to write to the TX channel and read from the RX channel as shown.

```
1 #pragma OPENCL EXTENSION cl_intel_channels : enable
2
3 channel float8 ch_eth_in __attribute((io("kernel_input_ch0")));
4 channel float8 ch_eth_out __attribute((io("kernel_output_ch0")));
5
6 __kernel void __attribute__ ((max_global_work_dim(0)))
7 sender(int length, __global float8 * restrict input)
8 {
9
10     for(int i=0; i<length; ++i)
11         write_channel_intel(ch_eth_out, input[i]);
12
13
14 }
15
16 __kernel void __attribute__ ((max_global_work_dim(0)))
17 collector(int length, __global float8 * restrict output)
18 {
19
20     for(int i=0; i<length; ++i)
21         output[i] = read_channel_intel(ch_eth_in);
22
23 }
```

Add section to describe loop pipeline in nexted loops

Add section to describe Memory banking and benfits in OpenCL™ kernels

Chapter 3

Topologies for FPGA-to-FPGA communication

This chapter will describe in detail the topologies which can be setup to connect the FPGAs using the QSFP ports to perform point-to-point communication to each other. The first section of the chapter will introduce the possible topologies which are feasible with the Noctua 32 FPGA system. The second section will describe the prototypes which were developed to evaluate two of the topologies to verify the functionality and compare the topologies in terms of bandwidth capabilities specifically for the MIDG2 application.

3.1 Topologies

As the current available BSP for the Nallatech 520N boards only support serial point-to-point communication between the FPGAs over direct connections, the configurations to build a topology is limited by the number of ports which is 4. The four ports would allow one FPGA to communicate simultaneously with 4 other FPGAs. To extend the communication beyond this, the FPGAs can either communicate via MPI using the host processor or hop over the FPGAs via the shortest path. Considering these criterion four topologies are feasible which either use MPI or hops to extend the communication above the 4 nodes.

Terminologies

To make the understanding of the topologies clear, this section would introduce some terms which would be used to describe the topologies in the next sections. The figure 3.1 shows a network of two FPGAs. The FPGAs act as the nodes in this network which are connected to each other with a direct link. To not confuse with the cluster node which are connected to each other using 100 Gb/s Intel Omni Path, the thesis will refer these nodes as FPGA and cluster nodes as node in rest of the text. A network in which all the nodes are connected to each other with direct FPGA-to-FPGA link will be called an *Isle*. The communication between isles is either done using hops or MPI via host. The nodes within the isle can either be fully connected or partially connected to each other.

3.1.1 Within Node

The simplest topology possible is to connect the FPGAs of a node to each other using a single channel or all the four channels forming an isle of two nodes as shown in the figure 3.2. The FPGAs can only communicate to each other directly over the channel(s) utilizing the complete bandwidth of the channels. The topologies can be scaled by adding more isles which communicate to each other using MPI via the host using the Intel Omni Path. The topology is simple and easy

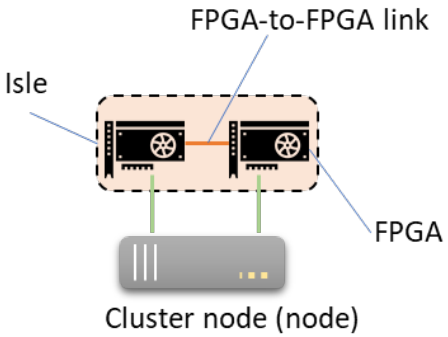


Figure 3.1: Simple network showing the network components

to setup. Applications which have large amount of data which needs to be transferred between the processes can benefit from this topology by efficiently partitioning and distributing the data among the isles and FPGAs.

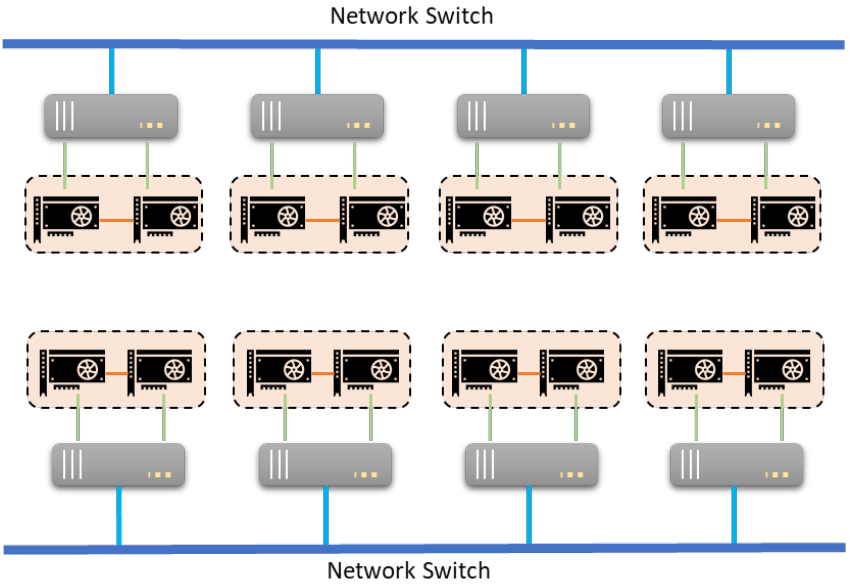


Figure 3.2: Within Node topology with two node per isle

3.1.2 Fully connected

The second topology extends the single node topology to two nodes such that an isle contains four nodes fully connected to each other with separated point-to-point link as shown in figure 3.3. Each FPGA in this topology can communicate with three other FPGAs simultaneously. Scaling the topology to more nodes can be achieved in two ways. The first way is similar to within node where the isles communicate using MPI. In this design, to decrease the overhead of exchanging data via the MPI, the data should be collected on a single FPGA using the point-to-point link and then exchanged via MPI. On the receiving end the process can be reversed. Some more details of possible strategies to scale this design effectively would be discussed in section

add reference

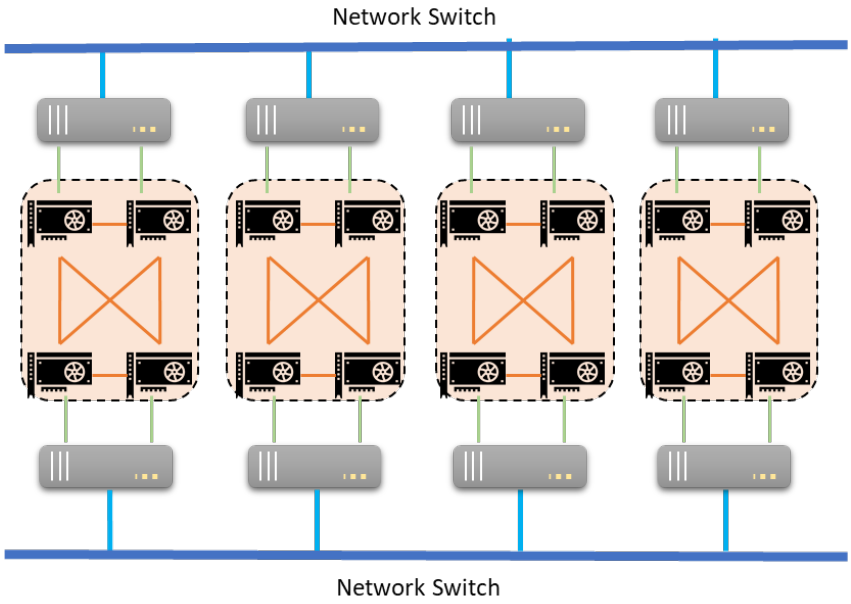


Figure 3.3: Fully connected topology of four FPGAs per isle

The second way to scale the design is to use the extra link left on the FPGAs to connect the isles to each other with two FPGA-to-FPGA links which is described in the section 3.1.3.

3.1.3 Connected Graph

This topology is an extension of the fully connected topology of the fully connected topology. The isle formed by the fully connected FPGAs is connected to each other using the fourth free port forming a connected graph network as show in figure 3.4. In this topology all the FPGAs can communicate to each other without requiring any data communication via host. In addition to the knowledge of fully connected mapping within the nodes, additional information about the neighboring isles would be required to be stored or configured in the FPGAs at compile time or at runtime. The additional information would be used by the FPGA to create a mapping table to route packets to the destination FPGAs via the shortest path. A data to be transferred from one FPGA to another in a different isle would then have to hop over the FPGAs along the shortest path to reach from source to destination.

This thesis proposes this topology as possible scaling design and there was no implementation and evaluation done for this topology as part of the thesis.

3.1.4 Toroidal

The last feasible topology for the FPGA network is the toroid. As explained by Robertazzi [19], A two dimensional toroidal network is a network in which the nodes on the left and right boundaries and the nodes on the top and bottom boundaries are connected to each other giving a fully connected network. The length and breadth of the network can vary depending upon the application requirements. As the maximum number of connection for per node required in the toroidal network is 4, the toroidal suits a lot to create a fully connected network of the FPGAs.

As Noctua has 32 FPGAs, two 4 X 4 torus as shown in figure 3.5 would be appropriate for connecting the FPGA giving an equidistant hops in each direction. The actual routing and

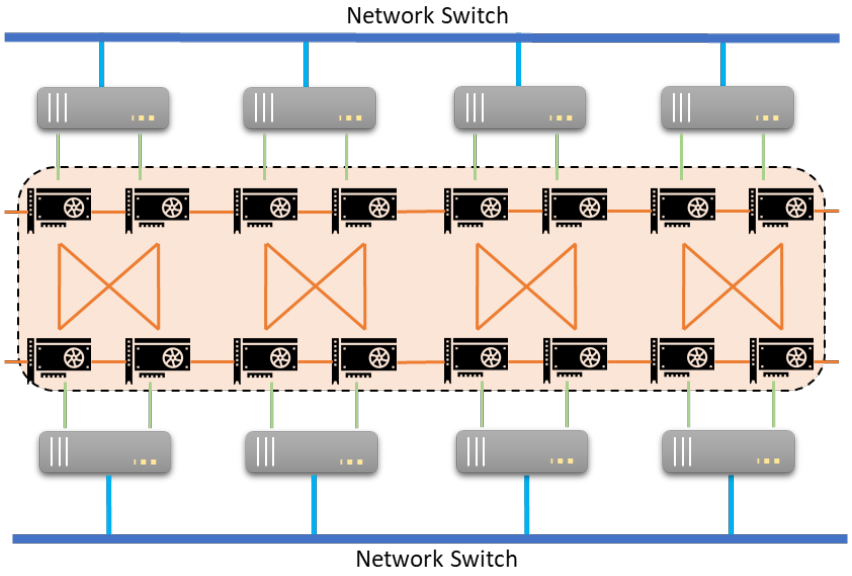


Figure 3.4: Connected Graph with HOPs between fully connected groups

packet forwarding strategies were not investigated in this thesis due to higher complexities and lack of hardware resources to achieve a result as part of the thesis.

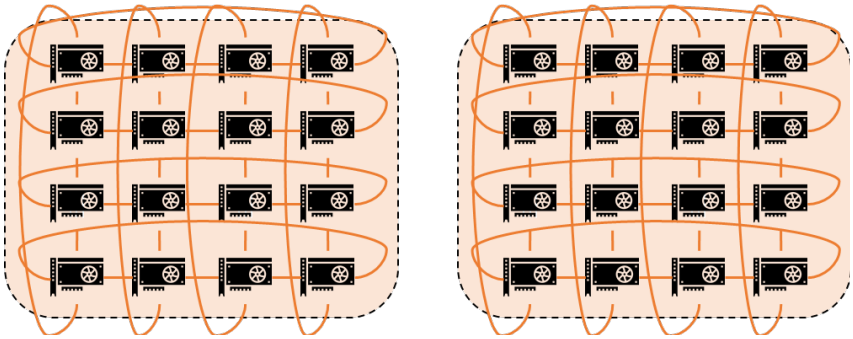


Figure 3.5: Two Toroidal network to connect 32 FPGAs

3.2 Prototypes to evaluate topologies

This section would describe the prototypes developed to evaluate the topologies introduced in sections 3.1.1 and 3.1.2.

3.2.1 Prototype for Within Node

The implementation of the within node topology was simple and required very few steps. To test the functionality and evaluate the achievable bandwidth with the within node topology two OpenCL kernels `sender` and `collector` were implemented. The code for the implemented kernels in shown in listing 3.1. `sender` kernel uses the `kernel_output_ch0` IO channel to send 256 bits of data per send. The data and the length of data to be transferred in each kernel execution is given by the host code through the parameters `input` and `length` respectively. The host copies

the data for `input` buffer into the FPGA global memory using `enqueueWriteBuffer()` API. The `collector` receives the data from the IO channel `kernel_input_ch0` and writes into the global memory `output` which is then read by the host using `enqueueReadBuffer()` to complete the data exchange between the FPGAs.

Listing 3.1: Kernels for within node prototype

```
1 #pragma OPENCL EXTENSION cl_intel_channels : enable
2
3 channel float8 ch_eth_in __attribute((io("kernel_input_ch0")));
4 channel float8 ch_eth_out __attribute((io("kernel_output_ch0")));
5
6 __kernel void __attribute__ ((max_global_work_dim(0)))
7 sender(int length, __global float8 * restrict input)
8 {
9     for(int i=0; i<length; ++i)
10        write_channel_intel(ch_eth_out, input[i]);
11 }
12
13 __kernel void __attribute__ ((max_global_work_dim(0)))
14 collector(int length, __global float8 * restrict output)
15 {
16     for(int i=0; i<length; ++i)
17        output[i] = read_channel_intel(ch_eth_in);
18 }
```

The same kernels are run on both FPGAs creating pairs of `sender` and `collector` communicating over the channels as show in figure 3.6 in the full duplex communication mode.

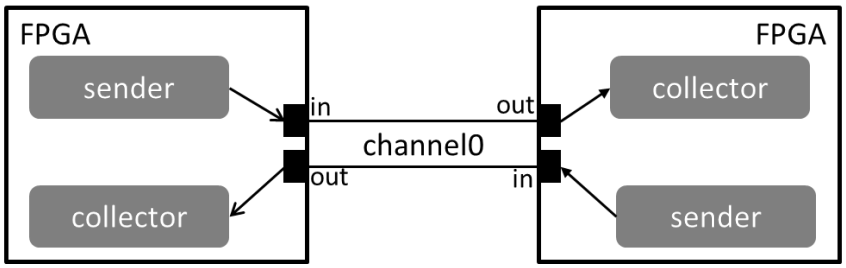


Figure 3.6: Communication structure of the kernels for within node topology with one channel

The host application was implemented using OpenCL C classes to reduce the amount of code and quickly test the functionality on the target platform. The host application is responsible for reading the synthesized binary files (`.aocx`) and use the `cl::Program` class to reconfigure the FPGA with the new binaries. The host application is also responsible to allocate the memories for the buffers `input` and `output` in host memory and in device memory using `cl::Buffer` class. The host code then sets all parameters for the kernels using the `cl::Kernel::setArg()` method and queues the kernels for execution on the FPGA using `cl::CommandQueue::enqueueNDRangeKernel()` method. The kernel used for prototype are implemented as a single work-item as it suits the serial IO channel communication.

Using all four channels for communication

Another prototype was also developed for the within node topology which uses all the four channels to communicate between the FPGAs. The benefit of this design is faster transfers for large data sizes as communication is performed on all the channels parallely giving higher data

rates. The modifications done to the kernels to use all the four channels for communicating is shown in listing 3.2

Listing 3.2: Kernels for within node using four channels

```
#pragma OPENCL EXTENSION cl_intel_channels : enable
channel float8 ch_eth_in0 __attribute((io("kernel_input_ch0")));
channel float8 ch_eth_in1 __attribute((io("kernel_input_ch1")));
channel float8 ch_eth_in2 __attribute((io("kernel_input_ch2")));
channel float8 ch_eth_in3 __attribute((io("kernel_input_ch3")));

channel float8 ch_eth_out0 __attribute((io("kernel_output_ch0")));
channel float8 ch_eth_out1 __attribute((io("kernel_output_ch1")));
channel float8 ch_eth_out2 __attribute((io("kernel_output_ch2")));
channel float8 ch_eth_out3 __attribute((io("kernel_output_ch3")));

__kernel void __attribute__ ((max_global_work_dim(0)))
sender_all(int length, __global float8 * restrict input)
{
    for(int i=0; i<length; i=i+4)
    {
        write_channel_intel(ch_eth_out0, input[i]);
        write_channel_intel(ch_eth_out1, input[i+1]);
        write_channel_intel(ch_eth_out2, input[i+2]);
        write_channel_intel(ch_eth_out3, input[i+3]);
    }
}

__kernel void __attribute__ ((max_global_work_dim(0)))
collector_all(int length, __global float8 * restrict output)
{
    for(int i=0; i<length; i=i+4)
    {
        output[i] = read_channel_intel(ch_eth_in0);
        output[i+1] = read_channel_intel(ch_eth_in1);
        output[i+2] = read_channel_intel(ch_eth_in2);
        output[i+3] = read_channel_intel(ch_eth_in3);
    }
}
```

3.2.2 Prototype for Fully Connected

The prototype for fully connected topology is an extension of the within node such that it uses additional 2 channels for communicating with two other FPGAs. The prototype uses known topology for communication which is derived from the fixed channel-FPGA pair formed by the actual connections. The connection between the FPGAs are shown in figure 3.7.

The FPGA implementation for fully connected topology also uses two OpenCL kernels `sender_all` and `collector_all` for communication. The kernels now use three channels which are shown in listing 3.3. Each of the channel is used to communicate with 3 different FPGAs simultaneously. For the prototypes, the host is designed to communicate same amount of data to each FPGA although the kernels support sending and receiving different data sizes from each channels. The host was limited to use same size to keep the prototype simple in the starting. The integrated implementation requires using variable size and a mechanism to identify and assign the sizes for each channel using partitioning information is created which allows configuring the size dynamically.

Listing 3.3: Channels used for fully connected topology

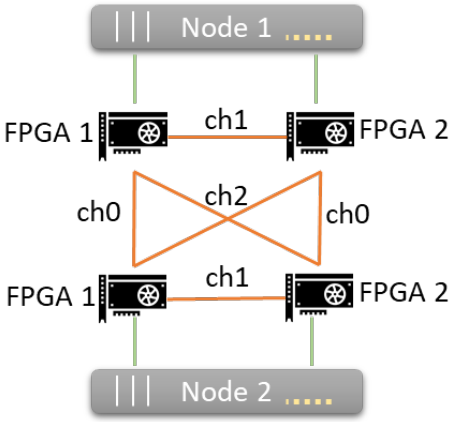


Figure 3.7: Hardware setup for the fully connected topology

```
1 #pragma OPENCL EXTENSION cl_intel_channels : enable
2 channel float8 ch_eth_in0 __attribute((io("kernel_input_ch0")));
3 channel float8 ch_eth_out0 __attribute((io("kernel_output_ch0")));
4
5 channel float8 ch_eth_in1 __attribute((io("kernel_input_ch1")));
6 channel float8 ch_eth_out1 __attribute((io("kernel_output_ch1")));
7
8 channel float8 ch_eth_in2 __attribute((io("kernel_input_ch2")));
9 channel float8 ch_eth_out2 __attribute((io("kernel_output_ch2")));
```

The implementation of the `sender_all` kernel is shown in listing 3.4. The kernel requires three separate memories and length, one for each of the channels, as parameters from host. The kernels iterates over the input buffers for the maximum length among the three lengths, sending 256 bits in each iterations over each channel. Individual lengths for each channel is used to stop the sends for the specific channel in following iterations. The structure shown in the listing creates three channel write units and three memory load Load store unit (LSU) which allows sending the data parallely on all the three channels in each iteration.

Listing 3.4: Sender Kernel for fully connected

```
1  __kernel void __attribute__ ((max_global_work_dim(0)))
2  sender_all(int length, int length1, int length2,
3             __global float8 * restrict input,
4             __global float8 * restrict input1,
5             __global float8 * restrict input2)
6  {
7      for(int i=0; i<length || i < length1 || i < length2; i++)
8      {
9          if (i < length)
10             write_channel_intel(ch_eth_out0, input[i]);
11
12         if (i < length1)
13             write_channel_intel(ch_eth_out1, input1[i]);
14
15         if (i < length2)
16             write_channel_intel(ch_eth_out2, input2[i]);
17     }
18 }
```

The `collector_all` used to receive data parallely from three FPGAs was implemented similar to `sender_all` kernel and requires same number of parameters as shown in listing 3.5. The kernel reads the data parallely from each of the channels and write them to the corresponding buffers using the respective lengths for each channel to limit the channel read.

Listing 3.5: Collector Kernel for fully connected

```
__kernel void __attribute__ ((max_global_work_dim(0)))
collector_all(int length, int length1, int length2,
              __global float8 * restrict output,
              __global float8 * restrict output1,
              __global float8 * restrict output2)
{
    for(int i=0; i<length || i < length1 || i < length2; i++)
    {
        if (i < length)
            output[i] = read_channel_intel(ch_eth_in0);

        if (i < length1)
            output1[i] = read_channel_intel(ch_eth_in1);

        if (i < length2)
            output2[i] = read_channel_intel(ch_eth_in2);
    }
}
```

All four FPGAs use the same kernels to communicate to each other over the channels. The structure of the communication is show in figure 3.8

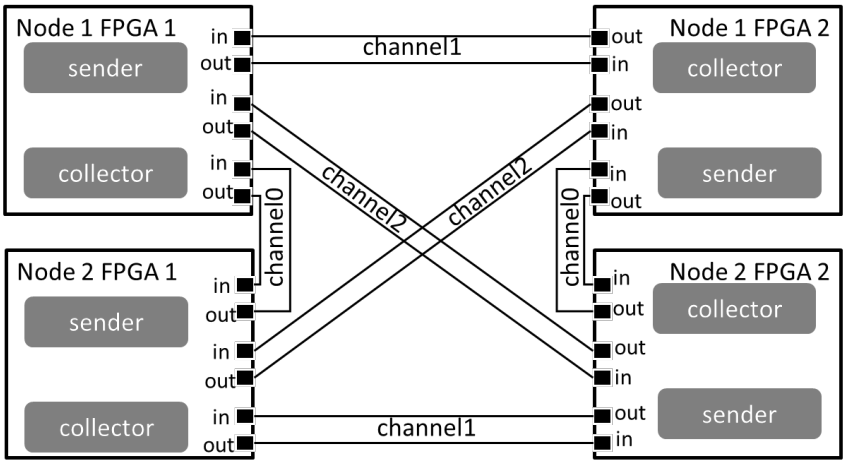


Figure 3.8: Communication structure for the Fully connected kernels on each FPGA

The implementation of the host application for the fully connected design additionally uses MPI. MPI is used to run the same host application on the 2 nodes. One each node the host application programs both of the FPGAs with the same kernel binary and initializes the kernel `sender_all` and `collector_all` on both FPGAs. Once the initialization is done, the host starts the kernel using the `cl::CommandQueue::enqueueNDRangeKernel()` method. The host waits for the completion of collector kernels on each FPGA, and then reads the received data for verification. Additionally `MPI_Barrier(MPI_COMM_WORLD)` functions is used the synchronize the hosts. This synchronization helps to reduce the stalls on the external channels.

Add the description of MPI PCIe prototype

Chapter 4

Integration of IO Channels in MIDG2

MIDG2[1] is used as the target application in this thesis to evaluate the benefits of using IO channels. MIDG2 application uses the DG method to calculate the electric and magnetic field values for objects as described in the chapter . The thesis extends a version of the application which uses OpenCL™ kernels to offload the performance critical calculations to the FPGA accelerators in a distributed system and use MPI to communicate among the different instances of the host application. The system scales over multiple nodes as discussed in section 2.2.1 but suffers from slow bandwidth due to the longer communication path over PCIe and Ethernet Network of host. To improve the bandwidth performance of the application, the IO channels are proposed to be used. This chapter will introduce and explain the changes which were done to the OpenCL™ kernels and the host application in order to use the QSFP Network Ports to build up topologies discussed in chapter 3 to allow direct communication between the FPGAs and reduce the communication time.

chapter number

4.1 Kernel Structure of MPI MIDG2 FPGA Implementation

The MPI MIDG2 FPGA version requires an additional `partial_get_kernel` to gather the shared elements from the main element buffer `g_Q` into a smaller buffer partial data buffer `g_partQ`. The main benefit of using the additional kernel is to avoid the large memory transaction between the host and the FPGA over PCIe to get the `g_Q` and assemble the shared buffers in host. The addition of the partial kernel was accompanied with resshuffling of the elements in the `g_Q` buffer. The distributed design requires the mesh to be partitioned into smaller meshes allocated to each rank. The elements of the partitioned mesh can either share a face with a neighboring rank or within the rank. The elements which share faces with neighboring ranks form the shared element group and the ones sharing faces within rank form the non-shared element group. To separate the computation of the these shared element group and the non-shared element group by reusing the same kernels required sorting of the elements into groups. This was done by placing the shared elements in the start of the `g_Q` buffer followed by the shared elements. This allowed to use the same kernels with different start and end element parameters to be invoked for processing the shared and non-shared elements separately.

add image to show the reaarangement to

The original pipeline developed for single FPGA was further optimized by introducing separate buffers `volrhsQ` and `surrhsQ` instead of the the same right hand side `rhsQ` buffer which was earlier accumulated in surface kernel. These changes allows the volume and surface kernel

[1] https://github.com/tcew/MIDG2

to execute parallely and write data into two separate buffers. The RK kernel reads both the buffers after volume and surface kernel are finished processing and accumulates the values using rungga-kutta constants to produce the field values for the timestep. Another improvement introduced to optimize the design was to use the concept of double buffers for the `g_Q`. The `g_Q` buffer is duplicated into two buffers `g_Q_ping` and `g_Q_pong` which use two alias `g_Q_in` and `g_Q_out` in the kernel. The `g_Q_in` serves as the buffer from which the kernels read the elements values for the current iteration wheres `g_Q_out` serves as buffer in which the values are updated. After each time step iteration the alias of the buffers is switched as shown in pseudo code in 4.1, exchanging the functionality of the buffers for the next iteration.

Listing 4.1: Buffer switching for double buffers in each iteration

```
1 for (itr = 0; itr < MaxTimeStep; itr++)
2 {
3     if (buffSwitch)
4     {
5         set_argument("g_Q_in", Q1_pong_mem);
6         set_argument("g_Q_out", Q1_ping_mem);
7     }
8     else
9     {
10        set_argument("g_Q_in", Q1_ping_mem);
11        set_argument("g_Q_out", Q1_pong_mem);
12    }
13    buffSwitch = !buffSwitch;
14 }
```

The concept helps to improve the performance of the RK kernel by removing read and write memory dependency on the same buffer. In each time step, the volume kernel and surface kernel first execute to compute the right hand side values for the non-shared elements while the shared data is exchanged. Once the shared data is available, the right hand side values for the shared elements is computed. After the computation, the RK kernels reads the field values from `g_Q_in` and `resQ` values for the last time steps and computes the accumulated field values using `volrhsQ` and `surrhsQ`. The computed values are then saved in `g_Q_out`. As the read and write buffers are separated, the memory dependency on the same buffer is reduced, reducing the latency to read and write to the memory. As explained, this structure also allows to overlap the communication via MPI with the computation of the fields for the non-shared data and reducing the effects of delayed communication. The kernel structure for base MPI MIDG2 FPGA is shown in Figure 4.1.

4.2 Kernel Structure with IO Channels

The prototypes developed for the topologies evaluation served as the basis to implement the support for IO channels in the OpenCL™ kernels for the MIDG2 application. The main modification required to enable the OpenCL™ MIDG2 kernels to communicate via IO channels was to remove the `partial_get_kernel` and replace with two kernels `partial_send` and `partial_recv`. The implementation of these kernels is similar to the prototype `send` and `recv` kernels described in section 3.2. Two different set of kernels for the two topologies, the within node using 2 FPGAs on the same node and fully connect topology using 4 FPGAs on two nodes was developed.

explain the channels implementation further with source code to highlight that, aliasing of the g__partQ buffer for fully connect design

An additional optimization to kernel pipeline was also introduced which helps to improve the

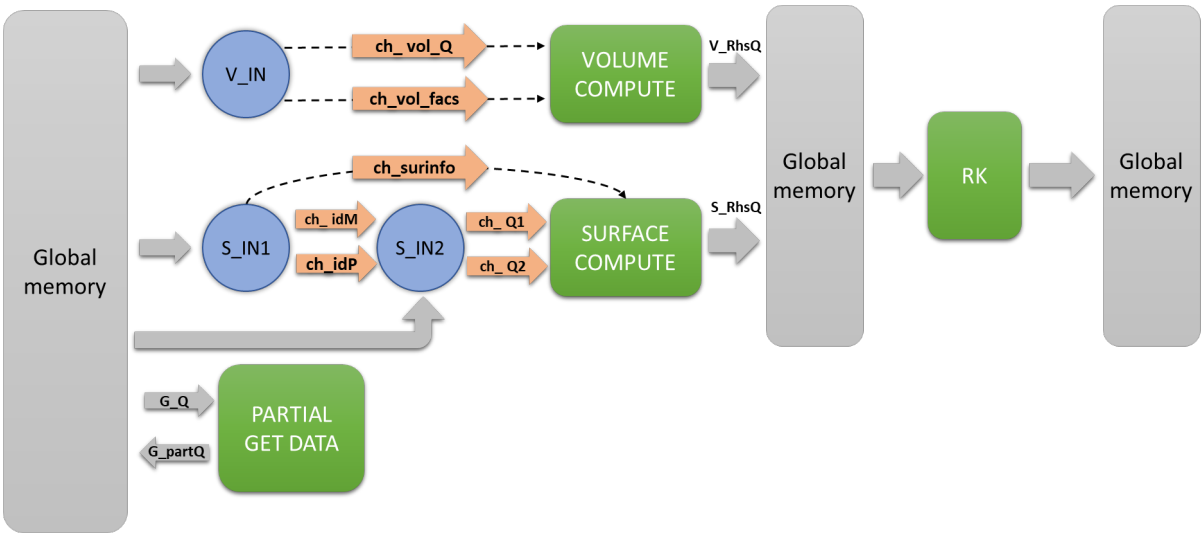


Figure 4.1: Structure for MPI FPGA OpenCL™ kernels showing the partial get kernel used to gather the shared elements into a separate buffer `g_partQ` for host to read

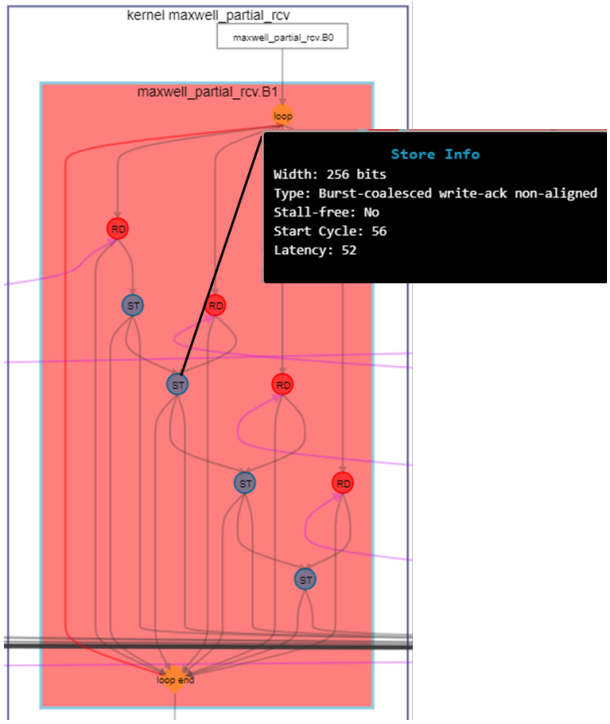


Figure 4.2: Memory dependency resulting in serial channel reads in the `partial_recv` for fully connected design

performance of the kernels marginally. Intel® OpenCL™ channels are used to communicate the right hand side field values from volume and surface kernels to the RK kernel as shown in the kernel structure in figure 4.3. Use of channels allows execution of all three kernels simultaneously giving a deeper pipeline structure and improving the throughput of the design by performing parallel computation in volume, surface and RK kernels.

As in the MPI MIDG2 FPGA design, the computation of shared elements follows the compu-

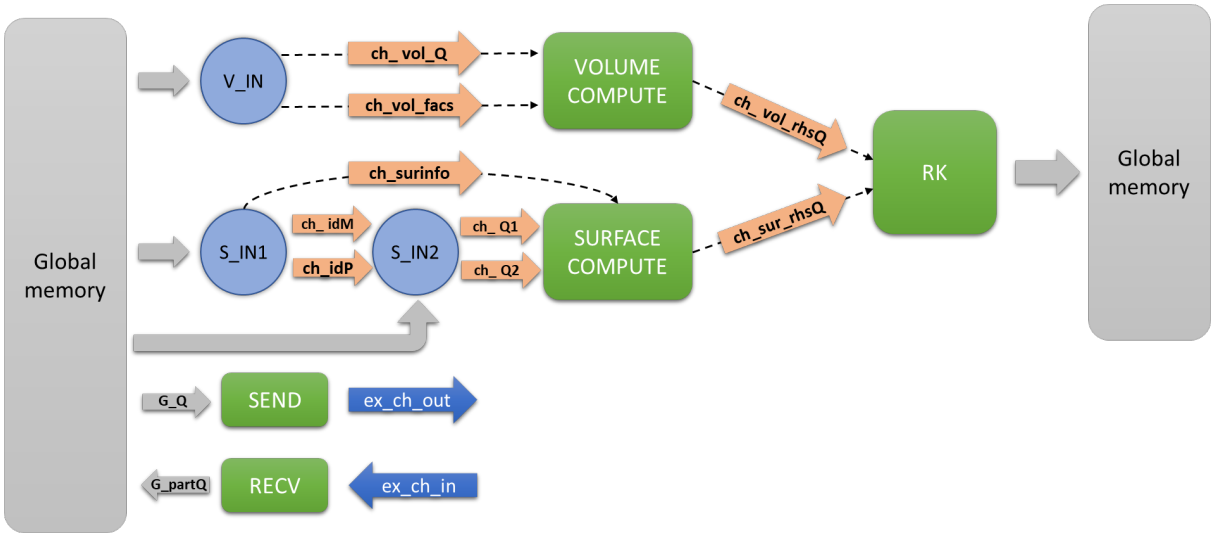


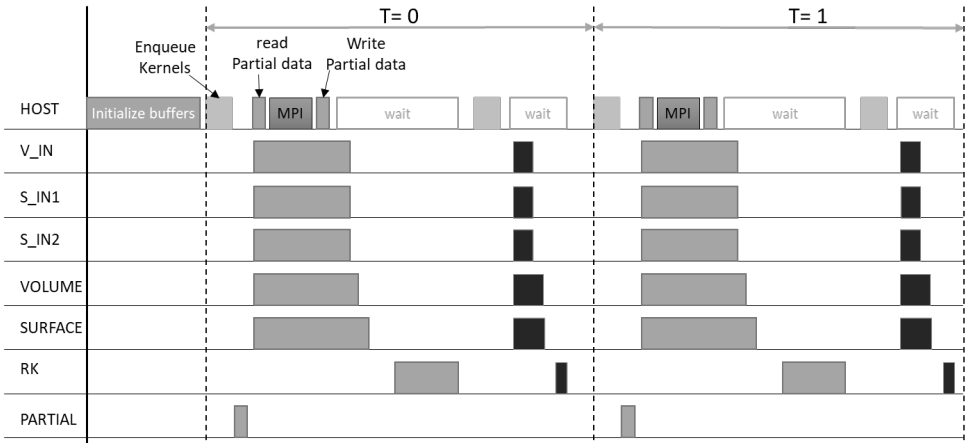
Figure 4.3: Structure for MPI FPGA OpenCL™ kernels showing `send` and `recv` kernels used for communication with external IO channels. Image also shows the internal channels introduced between volume/surface to rk kernel

tation of non-shared elements. The major change is the independence of the OpenCL™ kernels to communicate the data between the FPGAs without requiring support of the host for communication. Though host still controls the time step iteration and performs synchronization of kernels between the computation of non-shared data and the completion of communication of data between two FPGAs. A comparison of the sequence of operations performed on host and FPGA for the MPI MIDG2 FPGA and the IO channels design is shown in figure. As the figure shows, the involvement of the host is decreased. Also due to addition of the channels between the volume/surface and RK kernel, all kernels in the new design start together instead of RK following volume/surface.

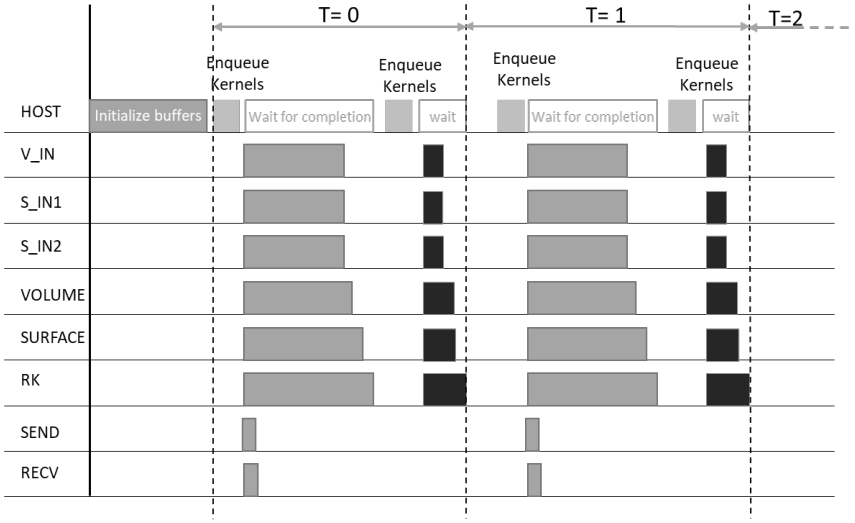
4.3 Host changes for to support IO Channels

The host MIDG2 implements the 3D DG-method introduced in the section 2.1. The application is responsible for following activities:

- Read in the mesh and parse the element coordinates and save them in VX, VY and VZ vectors
- For a distributed implementation using MPI, partition the mesh using parMETIS library and redistribute the elements as per the partition.
- Create the information of the shared elements which should be used to identify and map the indexes in the field buffer to be shared with the target MPI ranks. The mapping is used to read the values and create the partial data buffers to be communicated via MPI or FPGA-to-FPGA link
- Perform polynomial interpolation mapping of the geometric element vectors to the standard tetrahedral element coordinates using r, s and t values
- Compute the geometric coefficients and data layout information (`vmapM` and `vmapP`)
- Setup the OpenCL™ environment which includes identifying the OpenCL™ platform and device, create OpenCL™ memory buffers to be used, copy the data to OpenCL™ memory buffers, program the FPGA with the target OpenCL™ kernel binary and create OpenCL™



(a) MPI MIDG2 FPGA



(b) MIDG2 FPGA with IO channels

Figure 4.4: Sequence of Kernel and host event for MPI MIDG2 FPGA and MIDG2 FPGA with IO channels

queue to run the kernels.

- Run the OpenCL™ kernels and synchronize the execution over the computed timesteps
- Read the results value and compare the analytical and computed results to compute the error to estimate the accuracy of the implementation

The changes in the host application done mainly are towards enabling the new kernels to function correctly which involves setting up the OpenCL™ buffers and control the queueing of the kernels in the correct sequence. To handle this, the structure of the existing OpenCL™ initialization code sections is updated to handle both the designs with the same host code. A hierarchical structure for the OpenCL™ initialization code is implemented such that, the generic functionalities which is separated from the design specific buffer initialization and kernel enqueue sequence setup. The new structure of the host code is shown in figure and would be explained in the section refsec:hostcodeupdate.

mage of the code structure

The host application requires an additional json file to get the information about the kernels in the binary such as the kernel parameter information, dependencies among the kernels and

groups of kernels to be executed parallely. These information is used to configure the kernel buffers as well as setup a execution order tree which is used to enqueue the kernels in required order and setup other kernel to kernel synchronization using the events. The main benefit of using the configuration json file is to ease the setup effort for the dependent kernel. The subgroup structure of the kernels in IO channels design is shown in listing 4.2.

Listing 4.2: Kernel subgroups used in Multi FPGA design to enqueue kernels

```
"kernel_subgroups":
{
    "compute_inner":
    {
        "single_work_item": "true",
        "kernels":
        {
            "maxwell_partial_send":{},
            "maxwell_partial_rcv": {},
            "inputkernel_vol": {},
            "maxwellvolumekernel": {},
            "inputkernel": {},
            "maxwellsurfacekernel": {},
            "inputkernel1": {},
            "maxwellrkkernel": {}
        }
    },
    "compute_halo":
    {
        "single_work_item": "true",
        "kernels":
        {
            "inputkernel_vol": {},
            "maxwellvolumekernel": {},
            "inputkernel": {},
            "maxwellsurfacekernel": {},
            "inputkernel1": {},
            "maxwellrkkernel": {}
        }
    }
}
```

The `compute_inner` subgroup is responsible for computing the field values for the non-shared data and perform the data exchange between the FPGAs using the IO channels. As there is no dependencies listed among any of the kernels in the subgroup, each of the kernels will be executed simultaneously. The `compute_halo` subgroup performs the computation on the shared elements. As the same kernels are responsible for performing the computation on the shared and non-shared elements, the kernels are repeated in both the groups. The subgroups help to easily create a execution tree grouping the same kernels in different groups and order depending upon the implementation which can then be easily executed on the FPGA device.

Another important modification is done in the host application to get the information about the mapping of the MPI ranks to the FPGA devices on the node. This mapping is required for the fully connect topology to setup the correct offsets and number of elements for each of the

channels. The mapping information allows to associate a specific external channel with a MPI rank and hence the offsets within the `g_index` buffer which contains the index values in the `g_Q` buffer for the shared elements. To map the channels, additional information regarding the MPI rank and associated FPGA device is required. This information is shared using a structure which contains the MPI host id and associated FPGA device Id which is exchanged using MPI. After receiving the information about all the ranks, the external channel mapping is computed locally using a decision tree. The flow chart in figure shows the logic of the external channel assignment at each node.

lowchart to ex-
the selection

Chapter 5

Removing host dependency for optimization

The timestepping loops which are dependent on the size of the meshes are present in the host. Every iteration, the host uses the helper class `kernel_group` method `enqueue_NDRange()` to enqueue the subgroup of the kernels to perform the computation for that timestep and synchronize the kernel execution to sequentially execute the kernels on non-shared elements followed by shared elements after the completion of data transfers. This requires two host interactions per iteration, one each to queue each of the subgroup shown in listing 4.2. The introduction of the IO channels for communication between the FPGAs allows the FPGAs to communicate with each other without the need of host for performing the communication. Implementation of the synchronization between the communication kernels and compute kernels in the FPGA is a possibility now to ensure the sequential execution. This will allow to remove the one host interaction per iteration. Another possibility is to move the timestepping loops to the kernel and create a structure which only requires host to interact twice per application invocation and perform the synchronization within the FPGA.

As the host interaction is a time consuming operation and adds a small latency to every iteration due to the two interactions, the next optimization opportunity available due to introduction of IO channels was to remove the host dependency on the kernels and create a kernel structure which is able to run for the computed number of timesteps and synchronize the kernels to produce the final results. This chapter introduces the designs considered to achieve this explains the changes done in order to achieve FPGA only design. The chapter also discusses the issues identified during the optimization due to tool updates and changed kernel structure to highlight the areas for improvements in the final design.

5.1 Design considerations

To remove the host dependency in the FPGA kernels, the main change required was to move the timestep loops to the kernels such that they are iterate for the specified timesteps. Each kernel have the timestep loops added at the top level as shown in the pseudo code in 5.1. The complete modifications done to kernel codes would be explained in section 5.2. The actual timesteps are passed as parameter to kernels and are available at the run time. Making this change additionally required to create a synchronization scheme in the kernels to synchronize the communication kernels with the compute kernels and to synchronize the iteration execution in each of the kernels. This section will present the designs evaluated to identify the best possible design.

Listing 5.1: Pseudo-code of kernel showing additional timestep loops added for creating FPGA only design

```
__kernel void kernelName(__private int arg1,
                         __private int arg2,
                         __private int timesteps,
                         __global volatile float  *restrict buffer1,
                         __global volatile float  *restrict buffer2
                        )
{
    // Outer timestep loop
    #pragma max_concurrency 1
    for (int step = 0; step < timesteps; ++step)
    {
        // 5 RK steps
        #pragma max_concurrency 1
        for(int intrk = 0; intrk < 5; ++intrk)
        {
            // Old kernel code inside here
            // Process/Read/Write element
        }
    }
}
```

5.1.1 Synchronization using blocking Intel OpenCL™ channels

The first design implemented utilizes the blocking Intel OpenCL™ channels to communicate the synchronization events via the channels between the kernels. As shown in figure 4.3, the existing pipeline structure for the kernels is build up using the channels to separate the functionalities for data access into separate input kernels (`S_IN` and `V_IN` and computation kernels (`VOLUME` and `SURFACE`). The `RK` as explained before is responsible for accumulating as well as writing the data into the memory. This structure builds a pipeline where the data is fed at one end, processed and then written into the memory at the other end of the pipeline every iteration once for non-shared data and then for shared data after it is received from other nodes.

In order to maintain the correct order of the processing in the kernels, following synchronization needs to be achieved among the kernels.

1. At the completion of data processing for non-shared elements, the input kernels should wait for the communication to complete
2. The input kernels should wait for the last element to be processed and data written into the memory in the current iteration before moving into the next interaction
3. The communication kernel should start communication at the start of every new iteration and wait until the processing is finished before starting the communication for the next iteration

In the current design, host performs these synchronization by controlling the start of the kernels to process non-shared data along with communication kernels first. The hosts waits for kernels to finish the processing and communication using the `waitforcompletion()` method before starting kernels again to process shared elements. This sequence from the host point of view is shown in figure . Add sequence in

The first iteration of this design used 5 blocking channels to synchronize the above mentioned events as shown in figure 5.1. The arrows in the figure denote the read/write dependency of the kernel on the channels. Kernel from where the arrow starts is responsible for writing the data into the channel and the end kernel reads this data. The synchronization is achieved by the blocking nature of the channels. For example, to synchronize an event between two kernels, `kernel1` and `kernel2`, they are connected with a channel named `syncChannel`. Whenever `kernel1` should wait for a event from the `kernel2`, `kernel1` invokes a blocking read using

`read_channel_intel(syncChannel)` on the `syncChannel` between the kernels which makes `kernel1` block on the channel read. Once the event happens `kernel2` writes a token in the channel which unblocks `kernel1`. As the channel are uni-directional, the blocking nature is suited to achieve the synchronization required in the MIDG2 kernels.

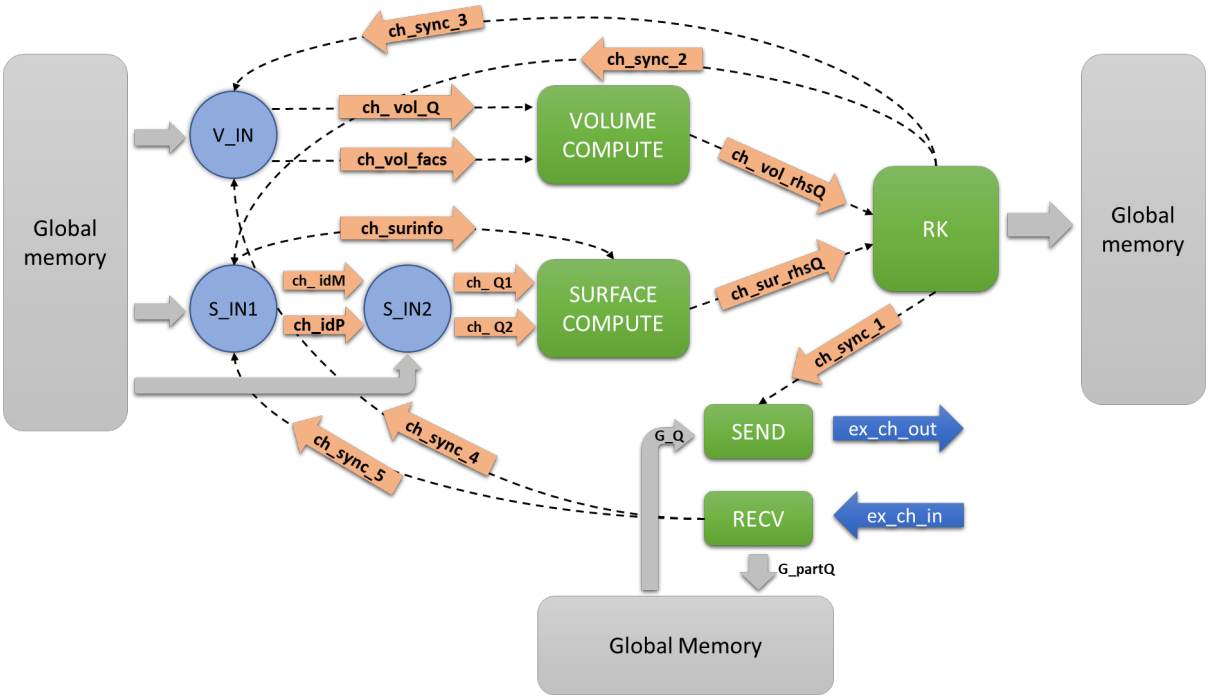


Figure 5.1: Kernel structure for FPGA only design utilizing 5 blocking channels for synchronization

In the design, `ch_sync_1` between RK kernel and the send kernel is to synchronize the start of communication at the beginning of each iteration. `ch_sync_2` and `ch_sync_3` is used to synchronize the completion of writing last element into the memory to start of reading the elements in the next iteration by the input kernels. `ch_sync_4` and `ch_sync_5` is used to synchronize the completion of communication from the `recv` kernels. Though the required sequence of operation was achieved with the channels and speed up was noticed, the design didn't produce correct results which was identified due to large deviation in the analytical and the computed nodal error values. After further analysis of the design it was noticed that the Load store unit in input kernels used cached access for `g_Q_ping` and `g_Q_pong` buffer reads which could be a problem with the updated design. As the buffer is updated by the RK kernel and the kernels are suppose to switch the buffers in each iteration, cached reads could lead to processing of stale values resulting in wrong field computation. To eliminate the cache for the memories, the buffer parameter were marked as `volatile` which as per the documentation allows to remove cached access as well as perform buffer management between the kernels to ensure correct sequence of reads and writes.

reference for

Addition of latency

Another probable issue with the design was with the difference in latency of memory operation and channel communication. As the channels are implemented as FIFOs in the hardware using registers or BRAMs, the latency of the channels is much lesser then that of a memory operation.

This would cause the input kernels to assess non-updated memory after receive of the event over the channels. As shown in the image 5.2, due to higher latency of the memory to handle the write request which is not visible in the kernel, an overlapped read is possible while using channels for synchronization.

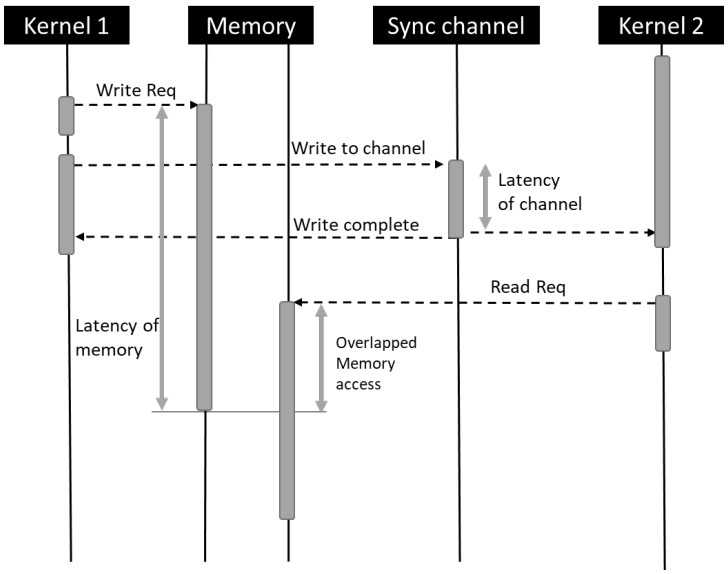


Figure 5.2: Sequence of operation between kernels using memory and channels showing the latency differences for memory and channel and its effect while using for synchronization

As there is no concrete information available for the target board to estimate the exact latency for the memory neither there is any method to block on the completion of the memory operation, it was not possible to fix this issue with a deterministic solution. Alternatively, a latency could be introduced after the invocation of last element write to ensure that the write memory operation complete before the read is invoked in the next iteration. As addition of latency in the OpenCL™ kernels in not trivial due to lack of standard APIs to add wait or sleep in the kernel, a latency logic was created using loops. A loop with an Initialization interval (II) of 1 can be used to create a latency for a desired amount of time by varying the loopcount. This is achieved by placing a channel write instruction in a loop as shown in the listing 5.2. The loop is pipelined with a II=1 is generated which is executed for `waitCount` iterations before writing the `token` into the channel. The `waitCount` is configured as a kernel parameter to vary the latency as per requirement as there is no easy deterministic way to measure the exact latency required. The latency in seconds added can be calculated using the frequency of the synthesized design using the formula

$$latency(inseconds) = \frac{waitCount}{Frequency(Hz)}$$

Listing 5.2: Loop structure used to add latency in the kernels

```
1 for(int time = 0; time < waitCount; ++time)
2 {
3     if (time == waitCount - 1)
4     {
5         write_channel_intel(channel, token);
6     }
7 }
```

Using this loop structure, latency was introduced in RK and `recv` kernels placing the channels `ch_sync_1`, `ch_sync_2` and `ch_sync_3` in a wait loop in the RK kernel and `ch_sync_4` and `ch_sync_5` in the another wait loop in `recv` kernel to avoid overlapped access of `g_Q` and `g_partQ` buffers. Introduction of latency changes the sequence of operations as shown in the figure 5.3 which shows no overlapping in the memory requests due to latency.

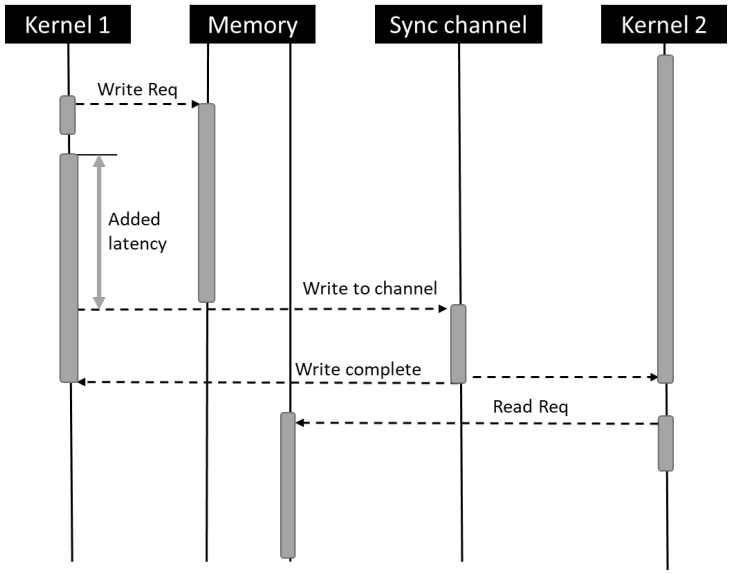


Figure 5.3: Sequence of operation with added latency to avoid overlapped memory access

Removing channels `ch_sync_2` and `ch_sync_3`

Another issue reported by the Intel OpenCL™ compiler as warning was formation of looping structure within the kernels due to use of the channels for synchronization. Addition of the `ch_sync_2` between RK and `S_IN1` kernel created a loop of the kernels `S_IN1` -> `S_IN2` -> `SURFACE` -> `RK` -> `S_IN1`. Similarly addition of `ch_sync_3` created a loop `V_IN` -> `VOLUME` -> `RK` -> `V_IN`. Creation of these loops avoided Intel OpenCL™ compiler to optimize the channel depths which caused increased latency of the pipeline and decreased performance.

After further analysis and tests with different kernel structure, it was identified that due to addition of the non-volatile memory for the buffers, an implicit synchronization of the memory operation would be created and the removal of channels `ch_sync_2` and `ch_sync_3` does not affect overall functionalities of the design in terms of speed and correctness. Due to this, it was decided to remove these channels which allowed the compiler to optimize the channel depth and improve performance.

5.1.2 Synchronization using locks with atomic memory operations

Apart from using channels for synchronization another possibility to achieve the desired behavior was by using atomic memory operations in the kernels to create an locking/unlocking structure for synchronization. Intel OpenCL™ FPGA SDK supports the standard OpenCL™ *Atomic Functions for 32-bit Integers* listed in 5.3.

Listing 5.3: Interger versions of the atomic operations supported by Intel OpenCL™ FPGA SDK

```
1 int atomic_add(volatile __global int *p, int val)
2 int atomic_sub(volatile __global int *p, int val)
```

```
 3 int atomic_xchg(volatile __global int *p, int val)
 4 int atomic_inc(volatile __global int *p)
 5 int atomic_dec(volatile __global int *p)
 6 int atomic_cmpxchg(volatile __global int *p, int cmp, int val)
 7 int atomic_min(volatile __global int *p, int val)
 8 int atomic_max(volatile __global int *p, int val)
 9 int atomic_and(volatile __global int *p, int val)
10 int atomic_or(volatile __global int *p, int val)
11 int atomic_xor(volatile __global int *p, int val)
```

The atomic operations were used to create a similar synchronization effect as done with the channels. `atomic_cmpxchg` and `atomic_xchg` functions are used to check and update the memory as shown in code listed in **??**. Four memory locations in the global memory are used for four synchronization behavior. The first and second location is to synchronize the access to g_Q memory between RK-volume and RK-surface pairs respectively. Volume and surface kernel wait for the RK kernel at start of every iteration to clear the memory to mark the start of new iteration. Immediately after clearing the memories, both write a specific token value to denote start. The third and fourth memory locations are to synchronize the start of communication and completion of communication. Send kernel waits for the memory location to be cleared by the volume and surface kernel at the start of every new iteration to start the communication. The same memory location is checked by volume and surface kernel after completion of processing the non-shared elements to have a specific token value written by the recv kernel after the completion of communication. This creates the synchronization between the start of communication to end of communication for processing the shared data.

Listing 5.4: Synchronization Implementation with atomic functions

```
 1 #pragma max_concurrency 1
 2 for (int step = 0; step < timesteps; ++step)
 3 {
 4     #pragma max_concurrency 1
 5     for(int intrk = 0; intrk < 5; ++intrk)
 6     {
 7         // Wait for memory 1 to be cleared
 8         while (atomic_cmpxchg(&lock[1], 0, 0xFB) == 0xFB);
 9
10         // Clear memory 3
11         atomic_xchg(&lock[3], 0);
12     }
13 }
```

As with channels, this design also had several similar issues. The use of atomic APIs are very expensive and reduce the clock frequency and overall performance of the design diminishing any benefit from the improved structure. Also like with channels, the memory latency resulted in incorrect results and required additional latency to be include. As the design was not good in terms of performance further analysis to fix the issues identified were not carried out due to timing constraints

5.2 Final optimized Design

The final design which is used for the evaluation and highlighting the benefits of the design includes options selected after multiple iterations of variations in the design in terms of kernel structure for synchronization, loop structure, variation in sequence of kernels and variation of the memory channel assignment to get the most optimized design as a whole. This section will present in detail individual aspect of the final design bringing the changes together in the kernel

as well as in the host code.

5.2.1 Kernel Structure

The final design with the selected optimization to maximize the performance of the kernel and achieve higher speed up was created using the design introduced in section 5.2.1. The final optimized design is shown in figure 5.4.

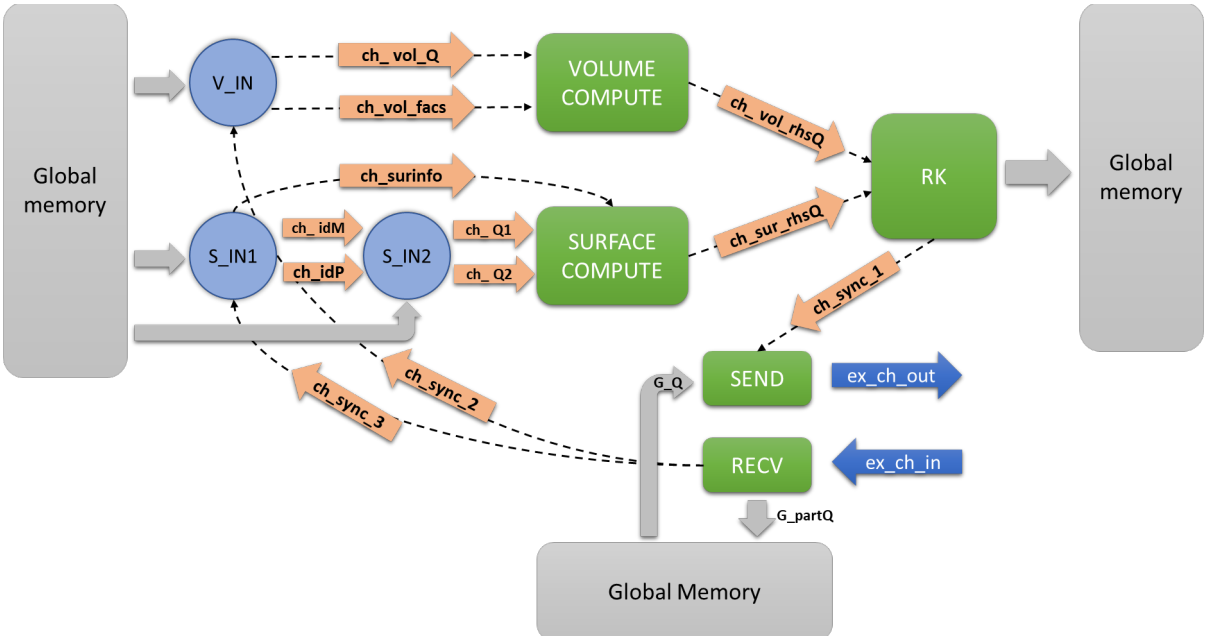


Figure 5.4: Kernel structure of the optimized FPGA only design

The kernels independent of host executes for `Nsteps` timestep iterations. In each timestep iteration 5 RK stages are executed. Within each RK stage, `K` elements are read processed and written back to the memory. The kernels form a pipeline where the k^{th} element and other coefficients required are read by the input kernels from the global memory and forwarded to the compute kernels by channels. The compute kernels process the element to compute right hand side field values and forwards it to RK kernel. There the field values are accumulated and stored in global memory. In between the shared data is communicated and used for computation. Due to the long pipeline, multiple elements are processed simultaneously in different stage of the pipeline giving higher throughput and performance.

The changes included in the final kernel structure are minor changes done to improve the performance and fix issues identified during testing of the different variations of the FPGA only implementation. Following points consolidate all the changes done to base design to achieve the final optimized design.

Loop coalescing

The major change from the base design introduced in section 4.1 to the optimized design is the introduction of the timestep loops in the kernels as shown in code 5.1. The introduction of these loops removed the dependency of the kernels on the host code for the timestepping but at the same time created the problem of increased nested loop depth. The increased nested loop depth is not ideal as it makes it harder for the Intel OpenCL™ compilers to optimize the loop pipelining. Another problem is the increase in the resource utilization as the additional control

hardware resources are required to handle each of the nested loops. To reduce the effects of this necessary change, in the final design, the top two levels of the nested loops were coalesced using the pragma `loop_coalesce` as shown in the kernel pseudo-code in 5.5. Use of `loop_coalesce` does not change the behavior of the actual loops instead only decreases the hardware resources required to handle the loops.

Listing 5.5: Loop coalescing used for additional timestep loops in FPGA only design

```
__kernel void kernelName(__private int arg1,
                         __private int K,
                         __private int timesteps,
                         __global volatile float  *restrict buffer1,
                         __global volatile float  *restrict buffer2
                        )
{
    // Outer timestep loop
    #pragma loop_coalesce 2
    for (int step = 0; step < timesteps; ++step)
    {
        // 5 RK steps
        for(int intrk = 0; intrk < 5; ++intrk)
        {
            for (int k=0; k < K; k++)
            {
                // Old kernel code inside here
                // Read/Process//Write k^th element
            }
        }
    }
}
```

Volatile memories and Buffer management

As it was identified during the analysis of the design with IO channels used for synchronization that the caching for memory was resulting in wrong results. The Memory buffers `g_Q1_ping`, `g_Q1_pong`, `g_Q2_ping`, `g_Q2_pong` and `g_partQ` are marked as volatile memory to avoid caching of the read LSUs. Along with the use of volatile memory, a buffer management scheme explained in the Buffer Management section in the Intel FPGA SDK for OpenCL Pro Edition: Programming Guide [1] was implemented to complement the synchronization via channels. The `mem_fence` with `CLK_GLOBAL_MEM_FENCE` and `CLK_CHANNEL_MEM_FENCE` flags were introduced after the buffer writes in the RK and `recv` kernels as explained in the document.

Communication Channels

The final design uses only three channels for synchronization as shown in the kernel structure in figure 5.4. The first channel is used between RK and `send` kernel to synchronize the start of the communication with the end of last iteration. The latency loop structure is used in the RK kernel which ensures that the `send` kernel reads the correct values from the `g_Q` buffer. Channels 2 and 3 connect from `recv` to `VOLUME` and `SURFACE` kernels to synchronize the end of the communication with the start of processing the shared elements. These channels writes are also within latency loop to add a delay for the read of the `g_partQ` buffer in surface kernel.

[1] https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807965224.html

Moving RK coefficients into kernel

Moving timestep loops into the kernels allowed an additional minor optimization in the design. The Low storage Runge-Kutta coefficients used for the accumulating the right hand side field values from `VOLUME` and `SURFACE` kernels were passed from the host application in each time step iteration. This was done from host as for each of the 5 Runge-Kutta stages a different coefficient is used and since the timestep and RK stage control was done in host application, the host selected the respective timestep-RK-stage coefficient and updated in the kernel private memory. Accessing the float coefficient add a small latency in each iteration of the execution and as the timestep loops were shifted to the kernels, the coefficients which are in total 10 float values are now stored in the kernel as constant arrays as shown in the code listing 5.6. This removes an additional memory access required to select the coefficient and improves the memory bandwidth and overall kernel performance.

Listing 5.6: Use of Runge-Kutta coefficients inside the RK kernel

```
1  __kernel void rkkernel(..Args..)
2  {
3      const float rk4a[5] = {0.0, -0.417890474499851962213366640245 39F,
       -1.192151694642676926105012355846 2F,  -1.697784692471527836226221789 3079F,
       -1.514183444257155781649068795 4744F};
4      const float rk4b[5] = {0.149659021999229117326491400117 37F,
       0.379210312999627280907498187323 67F, 0.822955029386981717170285373904 95F,
       0.699450455949122107038017583535 86F, 0.153057247968151992674142403319 02F};
5      // Outer timestep loop
6      #pragma loop_coalesce 2
7      for (int step = 0; step < timesteps; ++step)
8      {
9          // 5 RK steps
10         for(int intrk = 0; intrk < 5; ++intrk)
11         {
12             // Select the RK-coefficient for the stage
13             float l_fa = rk4a[intrk];
14             float l_fb = rk4b[intrk];
15
16             // Old kernel code inside here
17             // Process/Read/Write element
18         }
19     }
20 }
```

Buffer Aliasing

The change in the structure of the kernels due to introduction of the timestep loops required to move the buffer switching for the duplicated buffers `g_Q_ping` and `g_Q_pong` inside the kernels since host has no interactions with FPGA anymore during the timesteps. The input kernels and RK kernels receive both buffers as parameters and implement a switching logic as shown in the pseudo-code listing 5.7. The switching is done to switch the input buffer and the output buffer after every iterations which allows parallel execution of the all the kernels for computation.

Listing 5.7: Buffer switching for FPGA only design within the kernel

```
1  __kernel void kernelName(__private int K,
2                           __private int arg2,
3                           __private int timesteps,
4                           __global volatile float  *restrict g_Q_ping,
5                           __global volatile float  *restrict g_Q_pong
6                           )
```

```
7 {
8     // Outer timestep loop
9     #pragma loop_coalesce 2
10    for (int step = 0; step < timesteps; ++step)
11    {
12        // 5 RK steps
13        for(int intrk = 0; intrk < 5; ++intrk)
14        {
15            // compute timestep
16            int stepCount = step*5 + intrk;
17
18            // switch the buffer: Ping in EVEN step, Pong in ODD
19            __global volatile float *restrict g_Q_in = (stepCount%2 == 0)? g_Q_ping:g_Q_pong
      ;
20            __global volatile float *restrict g_Q_out = (stepCount%2 == 0)? g_Q_pong:
      g_Q_ping;
21
22            // Iterate over the elements
23            for (int k = 0; k < K k++)
24            {
25                float somevalue = g_Q_in[k];
26                ...
27                somevalue = somevalue + anothervalue;
28                ...
29                g_Q_out = someothervalue; // Creates Write-ACK LSU
30            }
31            // Old kernel code inside here
32            // Process/Read/Write element
33        }
34    }
35 }
```

The buffer switching using the above logic works correctly though resulted in false memory dependencies identifications in RK kernel by the compiler. As `g_Q_in` and `g_Q_out` are assigned the same buffers alternatively which compiler is unable to identify from the above structure and reports memory dependencies between the read and write memory operations which follow one another as shown in the pseudo-code 5.7. Due to the identified memory dependencies, the compiler adds the write-ack LSU for the writing operations which have higher latency and ensure that write operation is completed before any other operation on the same buffer is started. This is a false memory dependency which results due to inability of the compiler to identify the specified switching behavior. In order to solve this problem which resulted in decreased performance, memory aliasing is done for `g_Q_ping` and `g_Q_pong` memories similar to one done for the `g_partQ` memory. For aliasing, additional duplicate memory buffer parameters are assigned to kernel and used to assign the double buffers as shown in the pseudo-code 5.8. In the host same OpenCL memory buffer is assigned to both the parameters instead of creating additional memories which is not required. This fools the compiler to accept them as two different memories and avoid write-ack LSU. Similar approach was used with `g_resQ` memory also to avoid the same problem by aliasing the buffer into `g_resQ_in` and `g_resQ_out`.

Listing 5.8: Buffer switching for FPGA only design within the kernel

```
1 __kernel void kernelName(__private int K,
2                          __private int arg2,
3                          __private int timesteps,
4                          __global volatile float  *restrict g_Q_ping,
5                          __global volatile float  *restrict g_Q_pong,
6                          __global volatile float  *restrict g_Q_ping_2,
7                          __global volatile float  *restrict g_Q_pong_2
```

```
 8                          )
 9 {
10     // Outer timestep loop
11     #pragma loop_coalesce 2
12     for (int step = 0; step < timesteps; ++step)
13     {
14         // 5 RK steps
15         for(int intrk = 0; intrk < 5; ++intrk)
16         {
17             // compute timestep
18             int stepCount = step*5 + intrk;
19
20             // switch the buffer: Ping in EVEN step, Pong in ODD
21             __global volatile float *restrict g_Q_in = (stepCount%2 == 0)? g_Q_ping:g_Q_pong
       ;
22             __global volatile float *restrict g_Q_out = (stepCount%2 == 0)? g_Q_pong_2:
       g_Q_ping_2;
23
24             // Iterate over the elements
25             for (int k = 0; k < K k++)
26             {
27                 float somevalue = g_Q_in[k];
28                 ...
29                 somevalue = somevalue + anothervalue;
30                 ...
31                 g_Q_out = someothervalue; // Cached LSU
32             }
33             // Old kernel code inside here
34             // Process/Read/Write element
35         }
36     }
37 }
```

5.2.2 Host code updates

Addition of the FPGA only design required reworking of the host code in order to support all the designs variants with a single host code for performing the evaluation. Apart from the multi design support changes there are also other minor changes done in the code necessary to initialize the memories for the kernel buffers at aligned boundaries and some other changes. This section will discuss the changes in detail to give an understanding to the reader.

Restructuring of the configuration module

The major change in the host code is done to the module structure of the code responsible for handling the the OpenCL™ platform initialization, configuration and execution. With the addition of FPGA only design, three variants of the kernels are available viz. MPI MIDG2 FPGA (SingleFPGA), FPGA-FPGA communication using IO channels with host synchronization (MultiWithHost) and FPGA only design (MultiFPGAOnly). Initially different versions of the same configuration code was used by including or excluding design specific changes using preprocessor directives or by selecting different files for compilation. These intermediate solution allows the host code to support only single design and requires modifications and recompilation of the code every time to use other design. Additionally, the designs with IO channel communication also have two variants with minor configuration changes which also increase the complexity of the host code.

To build a simpler and understandable host code which is able to support all the three

designs in a single binary, the host code is restructured by introducing hierarchical structure for the OpenCL™ handling part. The structure of the modified host code is shown in figure . The restructuring does not modify the existing interface to the OpenCL™ configuration routine instead splits the configuration module into device specific and design specific configuration modules. The `BuildRunCLDevice` provides the generic interface for OpenCL™ device configuration and execution. The design specific run modules `RunSingleFpga`, `RunMultiWithHost` and `RunMultiFpgaOnly` performs the design specific memory configuration and setup to run the specific kernels on the device. The design specific run module is configured at the run time by providing the command line parameter `-d <DESIGN>` which allows a single host code to handle all the designs without requirement for a recompilation.

add image of th
structured mod

To achieve this restructuring additional structures were included which allows to group the variables into functionality based groups. One of the important structure which holds the function pointers to design specific routines is created as shown in listing 5.9. This structure is initialized with the design specific routines to handle device functionalities which include creation of OpenCL™ buffers, Initialization of kernel arguments, writing data into the device global memory and design specific cleanup. Each of the design specific modules initializes individual structures with the specific routines. The top level `BuildRunCLDevice` requests a pointer to this structure at the runtime using the `XXXX_getDeviceIntfHandlers()` function. The handlers for only the selected design is requested and used to configure the FPGA with the selected design kernels.

Listing 5.9: Structure to hold the design specific interface function pointers and initialization example

```
typedef struct
{
    void (*fnInitKernelArgs)(tClObjs* clObjs, Mesh* mesh, tChannelInfo* channelInfo,
                             char* kernelConfig);
    void (*fnFreeKernelMem)(tClObjs* clObjs);
    cl_int (*fnCreateBuffers)(tClObjs* clObjs);
    cl_int (*fnWriteToBuffers)(tClObjs clObjs, tKernelInitParams params);
    void (*fnRunKernel)(Mesh *mesh, tKernelRunParams* runparams, tProfileInfo* profileInfo,
                        tKernelInfo kernelinfo, tClObjs* clObjs);
    int (*fnGetKernelData)(tClObjs clObjs, float* c_resQ, float* c_Q,
                           tKernelInitParams params, Mesh *mesh);
} tclIntfHandlers;

static tclIntfHandlers intfHandlers =
{
    initKernelArgs,
    FreeKernelMem,
    createBuffers,
    writeToBuffers,
    runKernel,
    getKernelData
};

tclIntfHandlers* XXXX_getDeviceIntfHandlers(void)
{
    return &intfHandlers;
}
```

This structure allows the host application to request the design specific handlers at the runtime and support different designs in one binary. The modifications were useful in testing and evaluation of the designs.

Utilize all Memory channels of Nallatech 520N

memory image

The Nallatech 520N boards using Intel Stratix 10 FPGA provide 4 external memory channels to access the global memory as shown in the figure . Each of the 4 external memory channel can be used parallely by the kernels to read and write data into the global memory. Placing OpenCL™ buffers into different memory channels can improve memory bandwidth and overall performance of the kernels by reducing the stalls for simultaneous memory requests and reducing the latency of memory reads/writes. A OpenCL™ buffer can be configured in the host code to be placed in a specific channel by using the `CL_CHANNEL_X_INTELFPGA` flag during the creation of the memory as shown in the code below. `X` should be replaced with the channel number in which the buffer has to be placed. For Stratix 10 `X` can be from 1 to 4 as it supports 4 channels. Earlier Arria 10 boards supported only two channels.

```
1 cl_mem clMem;
2 clMem = clCreateBuffer(context, (CL_MEM_HETEROGENEOUS_INTELFPGA | CL_CHANNEL_1_INTELFPGA),
      size, NULL, &ret);
3 checkError(ret, "Failed to create buffer");
```

The MPI implementation was designed and optimized for the Nallatech 385A FPGA Accelerator Card which has a Intel Arria 10 GX FPGA. As the board supported only two channels, the host code used only two channels to distribute the memory buffers in the most optimal scheme. As there were additional two channels available on the Nallatech 520N board, the host code was updated for the base design to utilize all the 4 channels. The modifications improved the bandwidth performance of the base design as well as the overall execution time by small factors. The memory channel scheme for the base design was not suitable for the final design since all the kernels execute parallely as well as there are some memory buffers not used any more in the final design. The profile information of the final kernel design was used to come up with a different channel assignment scheme which distributed the memory load to all the four channels equally. The updated scheme uses channel 4 for `Q2 Pong` buffer as it was noticed to have stalls of upto 60% to 70% on writes to the buffer in the RK kernel. Stalls in the RK kernel are propagated over the complete pipeline of the kernels as all are connected via channels reducing the occupancy. Updating the channel reduced the memory stalls to 7% to 10% and improve the memory bandwidths and performance of the whole pipeline. Another change was to move the `resQ` to channel 3 instead of 2. As channel 3 was highly utilized due to removal of `volrhsQ` buffer RK kernel executes parallely The final channel assignment used for base design and the IO channels and FPGA only design is shown in table 5.1 and the definitions of the symbols used for size computation as below.

```
N = Order of the DG nodes
Np = ((N + 1) * (N+2) * (N+3)/6)
K = Number of elements
BSIZE = Np
Nfp = Number of DG nodes on each face of tetrahedral = (N+1) * (N+2)/2
Ntotalout = Number of shared fields
```

Alignment of the memory

The external channels support only 256 bits transfers. The first version using the IO channels for within the node topology used all the 256 bits to transfer data. This required aligning the partial memory buffer at 32 byte boundary to receive the data correctly. As the `partial_send` kernel requires to read the data from the `g_Q` buffer from random locations in as a group of 6 floats (24 bytes), use of 32 byte data transfers was not efficient for the non-aligned memory

Table 5.1: Points awarded to the evaluated tools

Buffer	Size	MPI	FPGA ONLY/IO CHANNEL
Q1 Ping	`K* BSIZE * pNfields`	1	1
Q1 Pong	`K* BSIZE * pNfields`	1	1
Q2 Ping	`K* BSIZE * pNfields`	2	2
Q2 Pong	`K* BSIZE * pNfields`	2	4
ResQ	`K* BSIZE * pNfields`	3	2
Mappping Info	`K* Nfp * NFaces * 2`	3	3
Surface Info	`K* Nfp * NFaces * 5`	3	3
partQ	`Ntotalout`	4	4
parMapOut	`Ntotalout`	3	3
vgeo	`12 * K`	3	3
surrhsQ	`K* BSIZE * pNfields`	3	-
volrhsQ	`K* BSIZE * pNfields`	4	-

access. To avoid this, the transfers were modified to send 24 bytes of actual data and pad the rest 8 bytes with 0s. This allowed to coalesce the 24 reads and improve the performance.

The similar approach was initially used for fully connected topology where 24 bytes of data on all 4 channels were transferred simultaneously. The send had no issues with this structured but as explained in section 4.2, the writes to `g_partQ` buffer were identified as dependent leading to a serial receives on the external channels and aliasing was used to solve this. The aliasing done requires the individual partitions of the shared memory to be aligned on a 64 byte as well as on a 24 byte aligned boundary. The 64 byte alignment is required to ensure that the parallel writes to each of the aliased buffer do not overlap due to cache line sharing. The cache is aligned on a 64 byte boundary and each non-aligned write should result in update of the whole cache line. The 24 byte alignment as the the reads from the buffer are at 24 byte aligned. To achieve this modifications were done in the `BuildMaps3d` function which collects the shared elements info and computes the indexes of the elements in the `g_Q` buffer. The individual partitions are aligned to a 192 bytes (48 floats) boundary which is the Least Common Multiple (LCM) of 24 and 64 using the code is 5.10. The code adds a padding in between the partitions ensures that the writes to the partitioned are not overlapped and also writes of 24 bytes are possible.

Listing 5.10: Alignment code introduced to ensure non-overlap writes of the aliased buffers

```
1  mesh->parNtotalout = 0;
2  int entries;
3  for (p2 = 0; p2 < nprocs; ++p2)
4  {
5      entries = skP[p2] * p_Nfields;
6      if (fFcdesign)
7      {
8          float value;
9          if (entries%16 != 0)
10         {
11             value = (float)entries/48;
12             entries = ceil(value)*48;
13         }
14         if (p2 != procid && entries > 0)
15             start[p2] = mesh->parNtotalout;
```

```
16              printf("[proc %d]: Start %d : %d: %d %f\n", procid, p2, start[p2], entries,
        value);
17      }
18
19      mesh->parNtotalout += entries;
20  }
```

Rearrangement of elements in g_Q

The FPGA only design processes all the K elements iteratively waiting for the shared data to arrive in between. As this requires the elements to be placed in the memory sequentially to be simplify and improve memory operations, the elements in the g_Q were sorted by arranging non-shared elements first followed by the shared elements. This is different from the initial sorting where the elements were arranged in the opposite order as shown in .

ref to the figure

5.2.3 Issues identified with optimized design

The FPGA only implementation was initially implemented with Intel FPGA SDK for OpenCL™ version 18.0.1 which was supported by the Nallatech BSPs. The new tool chain version 18.1.1 was release in mid of the thesis and had modifications which promised improvements to the kernel performance. It was decided to evaluate the design further with the newer tool chain versions to utilize any improvements for the Stratix 10 in the toolchains. The kernels were then compiled with the newer toolchain and issues were noticed in the kernel memory replication and banking. The compiler introduced arbitration units for the local memory buffers used in the VOLUME and SURFACE kernels as shown in the figure 5.5. This resulted in a decrease of performance due to stalled local memory reads which are expensive and causes stalls.

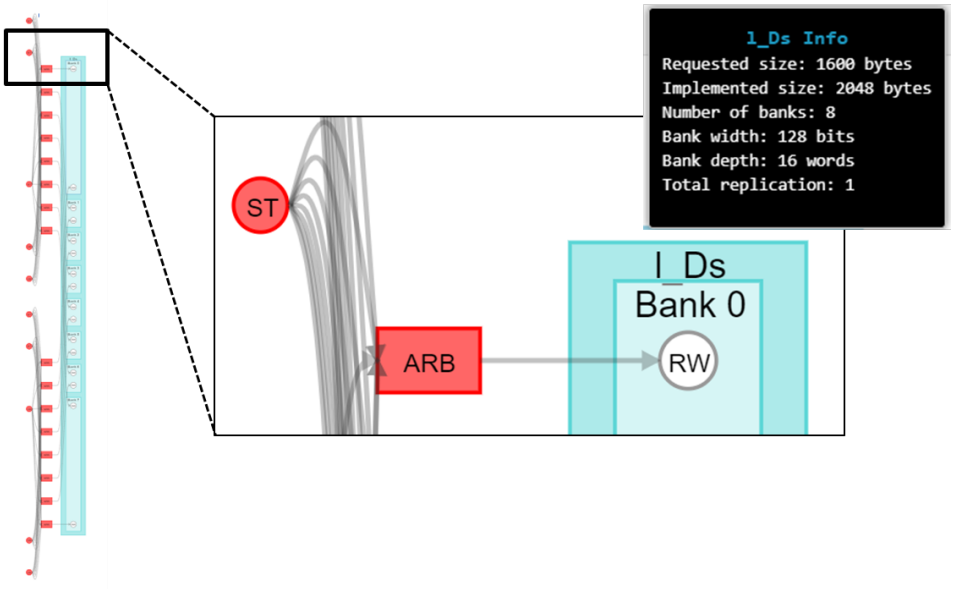


Figure 5.5: Local memory structure with arbitration units (ARB)

The kernels were analysed by looking at the synthesized reports and it was noticed that the 18.1.1 version of the compiler, auto unrolled the loops which were used to write data into the local memory from constant memory in both the kernels. The auto-unrolling created parallel store operations which resulted in the different banking and replication factor of the memories.

With different banking and replication factors, the compiler was not able to optimize all the memory accesses and produced stallable memory architecture for the local memories.

This was resolved by adding explicitly `pragma unroll 1` to the write loops, which prevented the duplication of store operations and restored the memory architecture as shown in figure 5.6

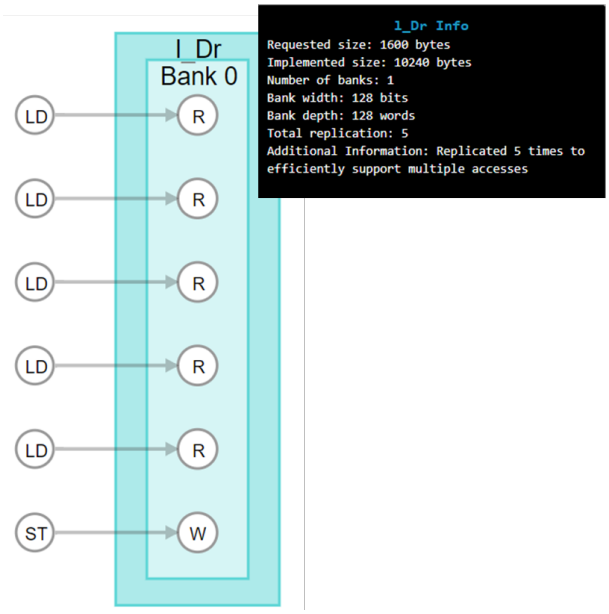


Figure 5.6: Local memory structure after addition with pragma unroll 1

Kernel pipelining issue

Chapter 6

Evaluation

6.1 Topology Evaluation

6.2 Comparsion with reference design

Bandwidth comparsion
: lower occupancy in the computation kernels giving hints to be computational bound

Chapter 7

Conclusion

List of Abbreviations

DG Discontinuous Galerkin
DGTD Nodal Discontinuous Galerkin Method in Time Domain
HPC high performance computing
II Initialization interval
LCM Least Common Multiple
LSU Load store unit
MIDG2 Mini Discontinuous Galerkin Maxwells Time-domain Solver
MPI Message passing interface
ODE ordinary differential equations
PCIe peripheral component interconnect express
PDE partial differential equations
SIMD Single Instruction Multiple Data

List of Figures

List of Tables

References

Literature

[1] Harold L. Atkins and Chi-Wang Shu. "Quadrature-Free Implementation of Discontinuous Galerkin Method for Hyperbolic Equations". *AIAA Journal* 36.5 (May 1998), pp. 775–782. URL: https://arc.aiaa.org/doi/10.2514/2.436 (visited on 03/11/2019) (cit. on p. 2).

[2] Abdalkader Baggag, Harold Atkins, and David Keyes. "Parallel Implementation of the Discontinuous Galerkin Method". English. Preprint. VA United States, Aug. 1999. URL: https://ntrs.nasa.gov/search.jsp?R=19990100667 (visited on 10/19/2018) (cit. on p. 2).

[3] Marc Bernacki et al. "Parallel discontinuous Galerkin unstructured mesh solvers for the calculation of three-dimensional wave propagation problems". *Applied Mathematical Modelling*. Parallel and distributed computing for computational mechanics 30.8 (Aug. 2006), pp. 744–763. URL: http://www.sciencedirect.com/science/article/pii/S0307904X0500212X (visited on 03/11/2019) (cit. on p. 3).

[4] K. Busch, M. König, and J. Niegemann. "Discontinuous Galerkin methods in nanophotonics". *Laser & Photonics Reviews* 5.6 (Nov. 2011), pp. 773–809. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/lpor.201000045 (visited on 03/05/2019) (cit. on pp. 2, 6).

[5] Bernardo Cockburn, Fengyan Li, and Chi-Wang Shu. "Locally divergence-free discontinuous Galerkin methods for the Maxwell equations". *Journal of Computational Physics* 194.2 (Mar. 2004), pp. 588–610. URL: http://www.sciencedirect.com/science/article/pii/S0021999103004960 (visited on 03/11/2019) (cit. on p. 2).

[6] G. Cohen, X. Ferrieres, and S. Pernet. "A spatial high-order hexahedral discontinuous Galerkin method to solve Maxwell's equations in time domain". *Journal of Computational Physics* 217.2 (Sept. 2006), pp. 340–363. URL: http://www.sciencedirect.com/science/article/pii/S0021999106000131 (visited on 03/11/2019) (cit. on p. 2).

[7] Gary Cohen, Xavier Ferrieres, and Sébastien Pernet. "Discontinuous Galerkin methods for Maxwell's equations in the time domain". *Comptes Rendus Physique*. Electromagnetic modelling 7.5 (June 2006), pp. 494–500. URL: http://www.sciencedirect.com/science/article/pii/S1631070506000740 (visited on 10/18/2018) (cit. on p. 2).

[8] S. Scott Collis. "Discontinuous Galerkin Methods for Turbulence Simulation". English. In: *Studying Turbulence Using Numerical Simulation Databases - IX: Proceedings of the 2002 Summer Program*. Stanford Univ.; Center for Turbulence Research; Stanford, CA United States, Dec. 2002, pp. 155–167 (cit. on p. 2).

[9] Michael Dumbser and Martin Käser. "An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes – II. The three-dimensional isotropic case". en. *Geophysical Journal International* 167.1 (2006), pp. 319–336. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1365-246X.2006.03120.x (visited on 03/11/2019) (cit. on p. 2).

[10] J. S. Hesthaven and T. Warburton. “Discontinuous Galerkin methods for the time-domain Maxwell’s equations”. *ACES Newsletter* 19 (2004), pp. 10–29 (cit. on p. 5).

[11] Jan S. Hesthaven and Tim Warburton. *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*. Texts in applied mathematics 54. OCLC: ocn191889938. New York: Springer, 2008 (cit. on pp. 2, 5, 6).

[12] Martin Käser and Michael Dumbser. “An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes – I. The two-dimensional isotropic case with external source terms”. en. *Geophysical Journal International* 166.2 (2006), pp. 855–877. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1365-246X.2006.03051.x (visited on 03/11/2019) (cit. on p. 2).

[13] Martin Käser et al. “An arbitrary high-order Discontinuous Galerkin method for elastic waves on unstructured meshes – III. Viscoelastic attenuation”. en. *Geophysical Journal International* 168.1 (2007), pp. 224–242. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1365-246X.2006.03193.x (visited on 03/11/2019) (cit. on p. 2).

[14] T. Kenter et al. “OpenCL-Based FPGA Design to Accelerate the Nodal Discontinuous Galerkin Method for Unstructured Meshes”. In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Apr. 2018, pp. 189–196 (cit. on pp. 3, 6–8).

[15] A. Klöckner et al. “Nodal discontinuous Galerkin methods on graphics processors”. en. *Journal of Computational Physics* 228.21 (Nov. 2009), pp. 7863–7882. (Visited on 10/19/2018) (cit. on p. 3).

[16] Ryohei Kobayashi et al. “OpenCL-ready High Speed FPGA Network for Reconfigurable High Performance Computing”. In: *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. HPC Asia 2018. New York, NY, USA: ACM, 2018, pp. 192–201. (Visited on 10/09/2018) (cit. on p. 3).

[17] Michael König, Kurt Busch, and Jens Niegemann. “The Discontinuous Galerkin Time-Domain method for Maxwell’s equations with anisotropic materials”. *Photonics and Nanostructures - Fundamentals and Applications*. Tacona Photonics 2009 8.4 (Sept. 2010), pp. 303–309. URL: http://www.sciencedirect.com/science/article/pii/S1569441010000246 (visited on 03/11/2019) (cit. on p. 2).

[18] W. H. Reed and T. R. Hill. *Triangular mesh methods for the neutron transport equation*. English. Tech. rep. LA-UR-73-479; CONF-730414-2. Los Alamos Scientific Lab., N.Mex. (USA), Oct. 1973. URL: https://www.osti.gov/biblio/4491151-triangular-mesh-methods-neutron-transport-equation (visited on 03/11/2019) (cit. on p. 2).

[19] T. G. Robertazzi. “Toroidal networks”. *IEEE Communications Magazine* 26.6 (June 1988), pp. 45–50 (cit. on p. 15).

[20] J. Sheng et al. “HPC on FPGA clouds: 3D FFTs and implications for molecular dynamics”. In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. Sept. 2017, pp. 1–4 (cit. on p. 3).

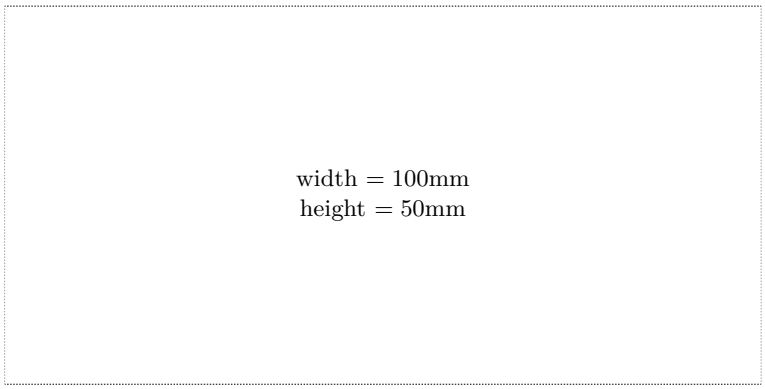
[21] Chi-Wang Shu. “Total-Variation-Diminishing Time Discretizations”. *SIAM J. Sci. Stat. Comput.* 9.6 (Nov. 1988), pp. 1073–1084. URL: https://doi.org/10.1137/0909073 (visited on 03/06/2019) (cit. on p. 6).

[22] Ioannis Toulopoulos and John A. Ekaterinaris. “High-Order Discontinuous Galerkin Discretizations for Computational Aeroacoustics in Complex Domains”. *AIAA Journal* 44.3 (Mar. 2006), pp. 502–511. URL: https://arc.aiaa.org/doi/10.2514/1.11422 (visited on 03/11/2019) (cit. on p. 2).

[23] Lucas C. Wilcox et al. “A high-order discontinuous Galerkin method for wave propagation through coupled elastic–acoustic media”. *Journal of Computational Physics* 229.24 (Dec. 2010), pp. 9373–9396. (Visited on 10/19/2018) (cit. on p. 2).

Check Final Print Size

— Check final print size! —



— Remove this page after printing! —