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

MASTERARBEIT

Submitted towards partial fulfilment
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 27, 2019

Gaurav Kumar Singh

Contents

Abstract

FPGA based accelerators provide the flexibility to adapt to application characteristics and are becoming popular among the high performance computing community. To utilize multiple of these FPGA accelerators in a parallel application, FPGA-to-FPGA communication is required over the nodes. This is currently performed by creating multiple copies of the data to be shared between the FPGA and CPU. This causes increase in the latency of the communication between the FPGAs and affects the application performance. This can be improved by using direct point-to-point communication between the FPGAs. To evaluate the benefits of the point-to-point communication, this thesis extends the parallel FPGA implementation of the MIDG2 application to use 4 100/40/25/10G QSFP28 network ports in the BittWare 520N accelerator boards to perform serial FPGA-to-FPGA communication.

The thesis first evaluates the possible point-to-point topologies between FPGAs using the 4 ports by implementing prototypes for 2 different topologies. The evaluation shows that the point-to-point communication can achieve a 10 times higher communication bandwidth than the existing MPI+PCIe communication. The point-to-point communication also has a higher efficiency of 99.4% compared to 60% for the MPI+PCIe. The point-to-point communication is then implemented for `MIDG2 MPI FPGA` application which uses Discontinuous Galerkin method to solve Maxwell's equation in time domain using multiple FPGAs. The existing OpenCL kernels are extended to use the point-to-point communication with 2 and 4 FPGAs and an optimized OpenCL kernel design is implemented to eliminate overheads due to FPGA to CPU interactions. The evaluation of the extended designs show a 20% to 30% improvement in the execution time of the application allowing the FPGA design to achieve linear speedup for scaling over 2 and 4 FPGAs.

Chapter 1

Introduction

FPGAs offer the flexibility to design application specific hardware for different kinds of computational intensive problems. They are capable of providing high memory bandwidths by using wide data buses and pipelines. As they are reconfigurable, reusing them for different kinds of problems is easy and does not add huge overheads. Such benefits have increased the popularity of the FPGA based accelerators as an alternative to the GPUs in a heterogeneous High Performance Computing (HPC). A High performance cluster contains multiple individual systems known as nodes. Each node can contain multiple multi-core processors, along with accelerators such as GPUs and FPGAs. The nodes are connected to each other using high speed interconnects 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 Figure 1.1. The applications run parallel on multiple cluster node and utilize the high speed interconnects to communicate and exchange data. Using multiple nodes with the FPGA/GPU accelerators allow to speedup the execution by use of more computing resources.

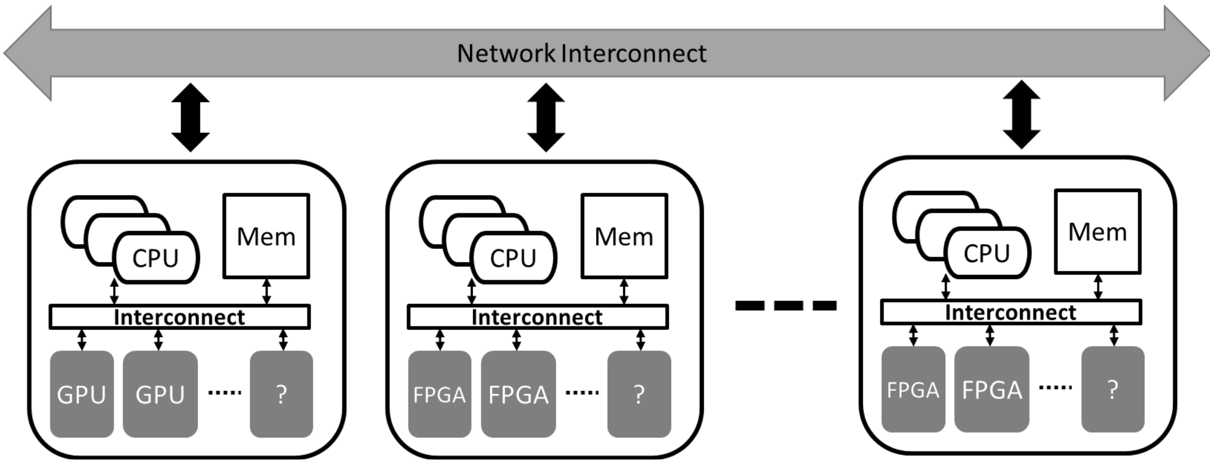


Figure 1.1: High performance cluster structure with multiple nodes having different configuration

The FPGA based accelerators in the nodes are mostly used to implement mathematical computations 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 them. Also, alternative hardware design development flows and tools are now used such as OpenCL which provide a C/C++ based design flow for the FPGAs. The OpenCL design flow is simpler and abstracts the complex hardware design challenges from a typical user of the High performance computing cluster making the application development with the FPGA accelerators simple.

To utilize the benefits of the FPGA accelerators, the new Noctua HPC cluster at Paderborn Center for Parallel Computing (PC^2) apart from having 256 compute nodes contains additional 16 nodes with 2 BittWare 520N FPGA accelerator boards having Intel Stratrix 10 FPGAs. Apart from having the high performance FPGA, the BittWare 520N boards are also equipped with four 100/40/25/10G QSFP28 network ports. These Network ports can be used as high speed serial IO channels to set up communication infrastructure between the FPGAs. Using the ports, the dependency of the FPGA on the CPUs for communicating the data can be removed. The high speed communication setup between the FPGAs can be used to scale up application over multiple FPGAs with small communication latency freeing the CPU to perform additional computation simultaneously.

MIDG2[1] is a Message Passing Interface (MPI) based parallel computation implementation of the DG [12] 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

Noctua HPC cluster equipped with FPGAs provides opportunities to create applications 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.

The two stages can be considered as individual sub-tasks. The first sub-task involves building prototypes using OpenCL IO channels which would utilize the four 100/40/25/10G QSFP28 network ports available on the BittWare 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 BittWare BSP for the channels. The second sub-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 are removed.

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

1.2 Related Work

DG method is believed to be first proposed by Reed and Hill [21] for solving the steady-state neutron transport equation [12]. 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, 25, 26], elasticity [9, 15, 16], electrodynamics Maxwell's equation [4–7, 20] 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 aeroacoustics scattering and uses MPI data exchange. Klöckner et al. [18] 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 utilize techniques such as efficient arithmetic operations using DSPs, deep instruction and pipelining to decrease computation time, multiple compute units 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. [17]. 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. While working at PC^2, We implemented an extension of this design before the start of this thesis which can use multiple single FPGAs to scale the application to 32 FPGAs using communication via CPU. This design is used as the base for this thesis.

The increased popularity of the FPGAs in past has led to identify possibilities for using multiple FPGAs. This can be achieved by using MPI communication via the CPU 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 low bandwidths and higher latency for the PCIe + MPI combination [19]. Systems have been investigated where the FPGAs could communicate directly without the need of host. Sheng et al. [23] 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 show 30% improvement over the reference design. Kobayashi et al. [19] 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 a 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 [19] but does not utilize the Ethernet-IP core and relies on the serial communication support provided in the BittWare BSP.

The rest of the thesis is divided into 5 chapters. Chapter 2 discusses the fundamentals components and technologies used in the thesis giving details of the DG method and overview of the OpenCL application development and techniques. It also introduces the single FPGA design presented in [17] and the distributed version using multiple FPGAs. Chapter 3 introduces the topologies which are possible for the FPGAs along with the prototypes developed. The chapter also presents the results of the evaluation of the topologies with synthetic data. Chapter 4 gives details of the first implementation done using IO channels describing the changes in OpenCL kernels. Chapter 5 presents the `FPGA only` system implemented by optimizing the IO channels design to remove the host dependency in the kernels. It also discusses the issues faced while optimizing the OpenCL kernels. Chapter 6 discusses the evaluation steps and the results of the evaluation comparing the design variants. It also presents a detailed analysis of the results to highlight the bottlenecks in the design. Final chapter 7 presents a conclusion of the achieved results of the thesis and proposes possible future works and improvements.

Chapter 2

Fundamentals and background

This chapter gives details of the various components used in implementation of the system described by this thesis. This will include the Software tools and techniques used to program the target design as well as the Hardware components essential to achieve the proposed topologies for the FPGAs. The chapter also introduces the base system and its background which is extended in the thesis.

2.1 Hardware and Software Platform

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

2.1.1 BittWare 520N

The BittWare 520N FPGA acceleration boards are used for the development and evaluation of the proposed system. The BittWare's p520_max_sg280l BSP provided with the board is used along with the board which provided support for

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 supported in the BSP as 4x40G Intel OpenCL external I/O channels
4. 16-lane PCI-Express Gen 3.0 for high speed host to FPGA data transfers (The current version of the BSP supports only 8 PCIe 3.0 lanes)

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 2.1 which shows the high computation capabilities of the FPGAs

QSFP Network Ports

The BittWare boards also contains 4 QSFP Network Ports which are connected to the FPGA transceivers to provide high speed network communication. As mentioned previously these ports

Table 2.1: Intel Stratix 10 FPGA GX2800 Resource summary

Resources	#
Logic elements (LEs)	2,753,000
Adaptive logic modules (ALMs)	933,120
M20K memory blocks	11721.00
M20K memory (Mb)	229
MLAB memory (Mb)	15
DSP	5,760

are used in this thesis to setup FPGA-to-FPGA networks to transfers data and reduce communication latency. The connections between the FPGAs are done using high speed fiber optic transceivers and cables capable of high speed long and short distance communications. The current version of the BittWare BSP supports 40 Gbits/s speed which is used for all calculation in this thesis.

2.1.2 Intel FPGA SDK for OpenCL

Intel FPGA SDK for OpenCL v18.0.1 Pro and v18.1.1 Pro is used for the development, debugging and synthesizing of the OpenCL kernels developed in this thesis. Intel MPI Library v18.0.3 is used for the MPI communication between the nodes for distributed setups. ParMETIS v4.0.3 is used for implementation of the partitioning scheme in the MIDG2 application.

The software development is based on the existing `MIDG2 MPI FPGA` code base which is described in section 2.3. The software package consists the `host application` which implements the mesh handling, partitioning, coefficient initialization and OpenCL platform initialization and control logic. The package also contains the OpenCL kernels which implement the computation logic of the DG method for distributed FPGA system using MPI+PCIe for communication between the FPGAs. This design is used as the base for the thesis and is extended in the thesis to include IO channels for communication in different topologies.

OpenCL application development

The OpenCL framework provides a programming environment for implementing applications for heterogenous systems which can include CPUs, GPUs, DSPs and FPGAs. An OpenCL application is divided into two parts `Host application` and `OpenCL KERNEL`. The `Host application` is executed on the `HOST` PC, mostly the CPU and the `OpenCL KERNEL` is executed on a separate hardware accelerator such as a GPU or FPGA. The `Host Application` can be implemented using C or C++ programming language and use the OpenCL runtime APIs to configure, control and execute programs on the target platform device. A `OpenCL KERNEL` is implemented using the OpenCL kernel programming language which is based on C/C++ and includes special programming constructs and keywords to use additional features for a target platform. The `OpenCL KERNEL` implements the computational logic which can be offloaded to the additional accelerator to speedup the computation.

For Intel FPGAs the `Host application` and `OpenCL KERNEL` are compiled separately. The `OpenCL KERNEL` code for FPGA is synthesized into binary using the Intel FPGA SDK for OpenCL offline compiler. `Host application` is compiled with the OpenCL runtime library and uses the OpenCL application runtime provided by the Intel to configure and execute the synthesized binary on the FPGA device.

Loop Pipelining and unrolling

The Intel FPGA SDK for OpenCL Offline Compiler optimizes the performance of the Single Work-Item kernels for data processing by creating efficient pipeline for the datapath of the loops. The pipelines can be used to start a new iteration of the loop per clock cycle if no data dependency between the iterations exists. This allows the reuse of the computation elements and parallel computation as multiple iterations simultaneously occupy the elements in different stage as explained in [**section 2.8.5**, 13]. Executing the loops in pipeline also allows the kernel to finish the computation faster.

Another performance improvement by increasing the parallelism can be done by unrolling the loop iterations. Intel FPGA SDK for OpenCL Offline Compiler allows to unroll loops iterations which creates more hardware resources to perform the operations in the loop iterations simultaneously with the additional resources. The compiler also tries to optimize the Load/Store operations in the unrolled loops by coalescing the memory operations to load or store all the data in single operation [**section 3.2**, 13].

Global Memory banks

The Intel FPGA SDK for OpenCL Offline Compiler uses the on board SDRAM as the global memory [**chapter 7**, 13]. The global memory is organized into multiple partitions and the number of available global memory banks depends on the FPGA board used. By default, the global memory is configured as burst-interleaved across the available memory banks. This configuration is suitable for memory access patterns which require sequential memory access of a very large single memory buffer accessed by a single kernel [**section 7.2.1**, 13].

The Intel FPGA SDK for OpenCL Offline Compiler also allows to configure the memory partitioning manually to place the buffers into different partitions depending upon the application needs. This is done by disabling the interleaving using the `-nointerleaving=<global_memory_type>` flag during compilation and allocating the buffer to one of the banks using the `CL_CHANNEL` flags [**section 6.2.1**, 14]. A design with multiple OpenCL `KERNEL`s accessing different buffers can benefit from this configuration because better load balancing will be achieved if the kernels access different memory banks simultaneously.

2.1.3 OpenCL Serial IO channels

The communication over the QSFP ports can be implemented using 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 BittWare BSP `p520_max_sg280l` provides 4 Tx and 4 Rx IO channels for the 520N board as shown in the Listing 2.2 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`.

Listing 2.1: IO channels description in `board_spec.xml`

```
<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"/>
```

Listing 2.2: IO channels usage example in a OpenCL kernel

```
#pragma OPENCL EXTENSION cl_intel_channels : enable

channel float8 ch_eth_in __attribute((io("kernel_input_ch0")));
channel float8 ch_eth_out __attribute((io("kernel_output_ch0")));

__kernel void __attribute__ ((max_global_work_dim(0)))
sender(int length, __global float8 * restrict input)
{
    for(int i=0; i<length; ++i)
        write_channel_intel(ch_eth_out, input[i]);
}

__kernel void __attribute__ ((max_global_work_dim(0)))
collector(int length, __global float8 * restrict output)
{
    for(int i=0; i<length; ++i)
        output[i] = read_channel_intel(ch_eth_in);
}
```

```
    <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 2.2 where two OpenCL kernels `sender` and `collector` are implemented to use the IO channels. The kernels use one TX (`ch_eth_in`) and one RX (`ch_eth_out`) channel to communicate in the `sender` and `collector` kernels. `ch_eth_in` and `ch_eth_out` are declared with `io` attribute 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.

2.2 Nodal Discontinuous Galerkin Method

The Nodal Discontinuous Galerkin Method in Time Domain (DGTD) [12] 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.

2.2.1 DGTD to solve Maxwell equations

Hesthaven and Warburton [11] 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 [12]. This section briefly describes 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 equation [12] 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

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

$\mathbf{H}$ and $\mathbf{E}$ are the magnetic and electric vector fields in three dimensions which are functions 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 an 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 [12] 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 for elements, the local solutions can be combined to give an approximation of the global solution. This is done by using an operator to combine the local solutions $u_h^k(x,t)$ elementwise except for nodal points on the faces. The nodal points on the faces of the tetrahedra are part of two adjacent D^k elements and have two different field values require coupling of the local solution. The coupling of the solution is done by computing the electric and magnetic field differences [17] $\Delta\mathbf{E} = \mathbf{E}^+ - \mathbf{E}^-, \Delta\mathbf{H} = \mathbf{H}^+ - \mathbf{H}^-$. The global approximate solutions are then computed from the local solutions [4, 17] by a pair of ordinary differential equations (ODE) for the semi-discrete system derivation which combines the local and global field values as explained in [12]:

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

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

The $(\mathcal{M}^k)$ 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 [24] is used [12, 17] 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.

2.3 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 2.2.1 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 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 Figure 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 by 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 2.2.1 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.

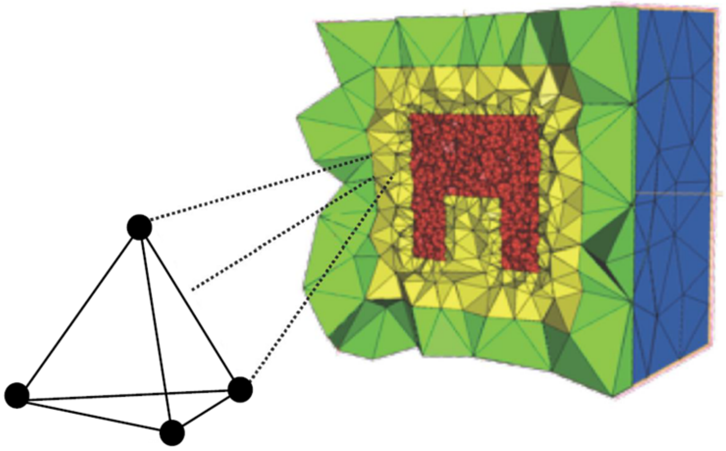


Figure 2.1: K element mesh with tetrahedra elements for split-ring resonator object taken from [10]

[1] https://libocca.org

[2] http://wias-berlin.de/software/tetgen/

The parallelization of 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 use an approximation operator to combine the local solutions element wise all Ks, 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 library 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 nodes or MPI ranks speedup the computation by distributing the computation load. The step wise process for partitioning and distribution is shown in Figure 2.2.

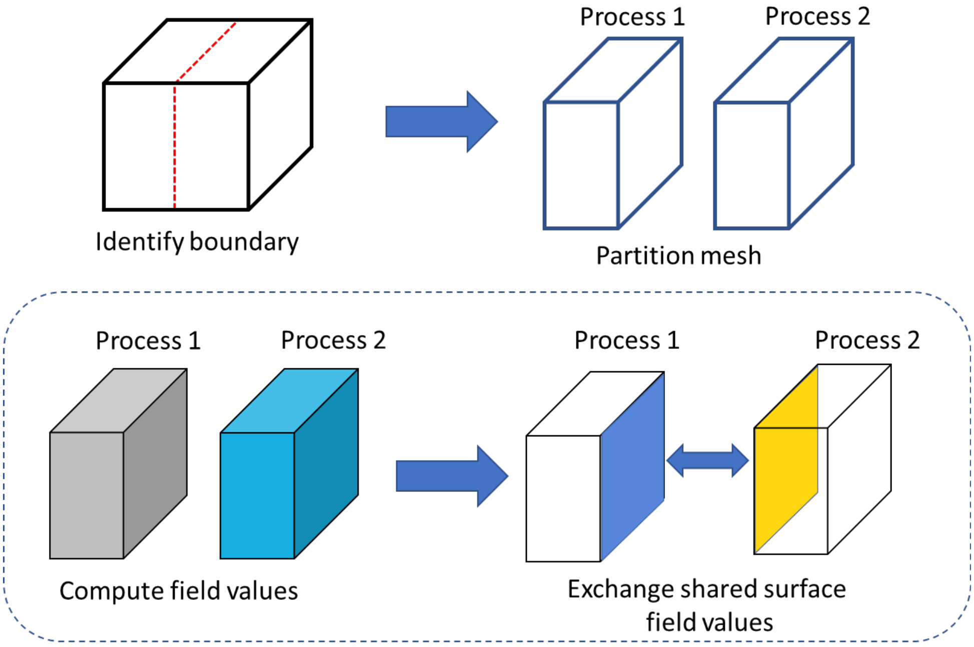


Figure 2.2: Parallelization achieved with partitioning and processing by multiple processes

2.4 FPGA implementation for DGTD (MIDG2 FPGA)

Kenter et al. [17] 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 multiple memory access channels to allow parallel data reads/writes for higher bandwidth with lower stalls. This section gives a brief introduction of the implementation.

[3] http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview

2.4.1 Kernel Structure

The structure of the OpenCL kernels for this implementation is shown in Figure 2.3 which shows pipeline structure developed for acceleration with the single FPGA. Separate IO kernels `V_IN`, `V_OUT`, `S_IN1`, `S_IN2` and `S_OUT` were added which uses the Intel OpenCL channels to feed in the data read from the global memory into the compute kernels as well as get the data from the kernels and write it into the memory. The use of channels helps to improve the memory performance as the OpenCL compiler automatically identifies the depths required for the channels to compensate the latency differences [17].

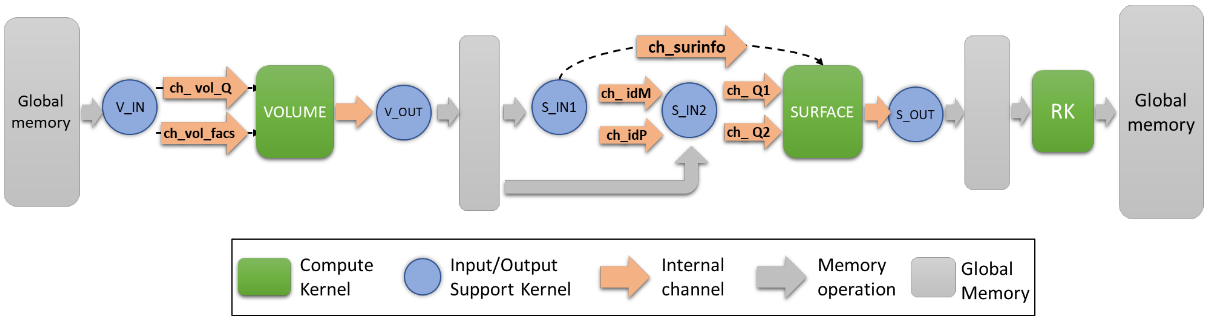
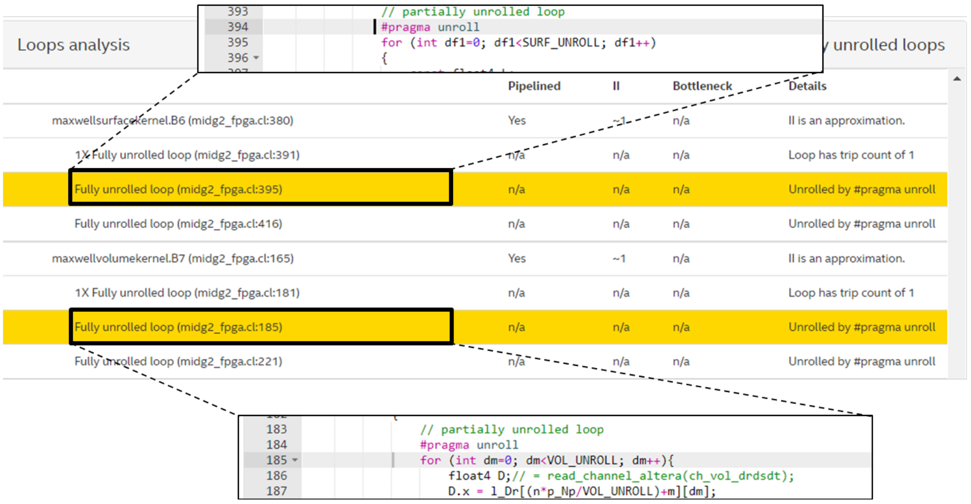


Figure 2.3: Block level structure for OpenCL kernels developed for single FPGA by Kenter et al. [17]

The kernels also exploit the benefits of using faster local memories in the kernels to store constants and data which can be reused over multiple iterations. The high degree replication of the floating-point operations on each element is achieved by fully unrolling the inner loops in `VOLUME` and `SURFACE` as shown in Figure 2.4



```
393                // partially unrolled loop
394                #pragma unroll
395                for (int df1=0; df1<SURF_UNROLL; df1++)
396                {
```

	Pipelined	II	Bottleneck	Details
maxwellsurfacekernel.B6 (midg2_fpga.cl:380)	Yes	~1	n/a	II is an approximation.
1X Fully unrolled loop (midg2_fpga.cl:391)	n/a	n/a	n/a	Loop has trip count of 1
Fully unrolled loop (midg2_fpga.cl:395)	n/a	n/a	n/a	Unrolled by #pragma unroll
Fully unrolled loop (midg2_fpga.cl:416)	n/a	n/a	n/a	Unrolled by #pragma unroll
maxwellvolumekernel.B7 (midg2_fpga.cl:165)	Yes	~1	n/a	II is an approximation.
1X Fully unrolled loop (midg2_fpga.cl:181)	n/a	n/a	n/a	Loop has trip count of 1
Fully unrolled loop (midg2_fpga.cl:185)	n/a	n/a	n/a	Unrolled by #pragma unroll
Fully unrolled loop (midg2_fpga.cl:221)	n/a	n/a	n/a	Unrolled by #pragma unroll

```
183            // partially unrolled loop
184            #pragma unroll
185            for (int dm=0; dm<VOL_UNROLL; dm++){
186                float4 D;// = read_channel_altera(ch_vol_drdsdt);
187                D.x = l_Dr[(n*p_Np/VOL_UNROLL)+m][dm];
```

Figure 2.4: Loop analysis report for the kernels showing fully unrolled inner loops of `VOLUME` and `SURFACE` kernels using `pragma unroll`. This replicates the floating-point units and perform parallel computation in the inner loops

The complete implementation details along with performance evaluation for the design is described in [17].

2.5 MIDG2 MPI FPGA implementation

The original MIDG2 implementation supports multi-node system with accelerators using kernels for GPU. These kernels perform the shared data communication via `Host Application` using MPI. Before the start of the thesis, as preparation phase, we extended the MIDG2 FPGA implementation to work on a multi-node cluster with multiple FPGA accelerators to create `MIDG2 MPI FPGA` design. The MIDG2 MPI FPGA design uses the same code base as the original MIDG2 MPI implementation along with the modifications done to support OpenCL framework for FPGAs. The mesh is divided into partitions using ParMETIS and the computation can be performed on different compute nodes using FPGA accelerators. The OpenCL kernels from the MIDG2 FPGA design are used to replace the original MIDG2 kernels for GPUs. An additional kernel which was used in the original MIDG2 implementation is included with necessary modification and used to read the shared data and store it in a coalesced memory. A detailed information of the updated kernel structure is given in section 2.5.1. The updated `host application` then reads this data from the FPGA global memory and uses MPI to transfer it to other nodes to which the data is shared. Figure 2.5 shows the system level architecture of this design for two nodes.

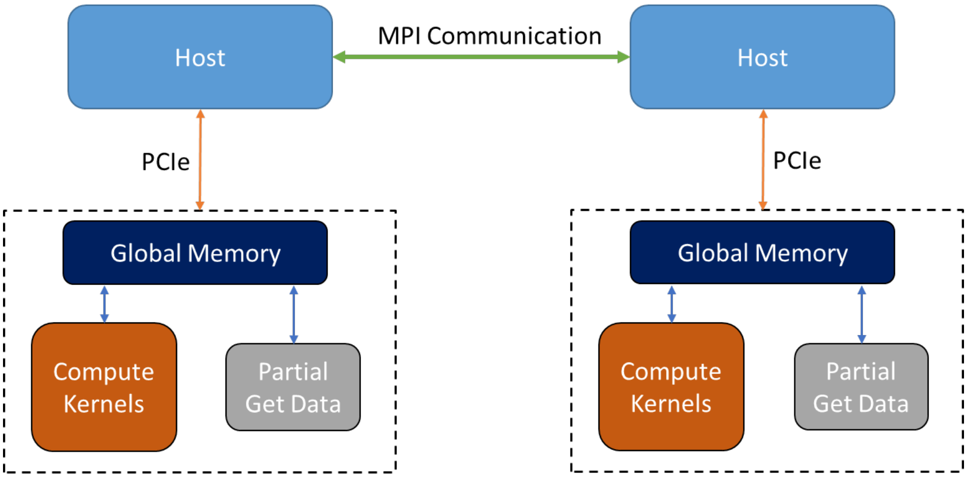


Figure 2.5: System level architecture for MPI FPGA design communicating using MPI and PCIe

Using this design, multiple FPGAs can be used to speed up the computation for large meshes. Though speedup can be achieved with this design, the communication which involves movement of data through PCIe bus and the network interconnect between the nodes, reduces the combined communication bandwidth a lot. This communication setup also requires having synchronization constructs between the FPGA kernels and the host application to ensure data correctness which has overheads and contribute to the increase of 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.

2.5.1 OpenCL Kernel Structure of `MIDG2 MPI FPGA` Implementation

The MIDG2 FPGA implementation works for a single FPGA and utilizes special OpenCL FPGA APIs and techniques to reduce the computation time for the electric and magnetic flux values as explained in section 2.4. In a distributed MIDG2 application the shared surfaces are identified after the mesh partition and a list of the shared nodes on the surfaces is prepared which is

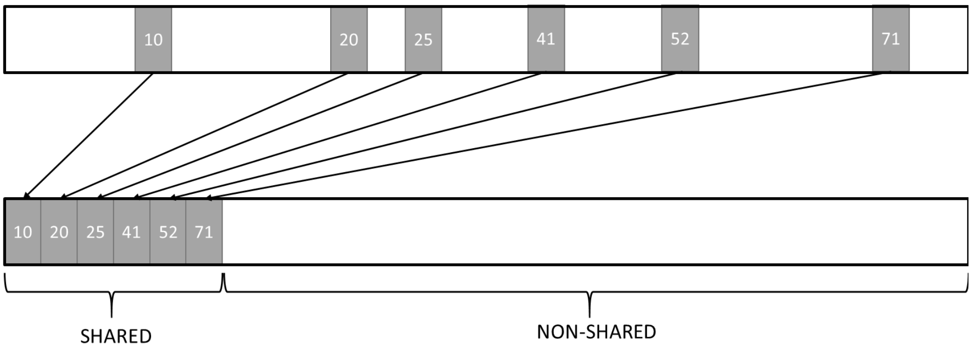


Figure 2.6: Rearrangement of `g_Q` buffer elements by sorting and bringing the shared elements in the beginning of the buffer

used to share the data. To exchange this shared data, MIDG2 FPGA kernels required a way to collect the shared data in the FPGA global memory which the `host application` could then read and exchange via MPI. To enable this, we added an additional `partial_get_kernel` kernel to the design 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 was 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 re-shuffling 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 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 non-shared elements as shown in Figure 2.6. 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.

Prior to start of the thesis, we further optimized the original pipeline developed in MIDG2 FPGA by introducing separate buffers `volrhsQ` and `surrhsQ` instead of the single right-hand side `rhsQ` buffer which was earlier accumulated in `SURFACE` kernel. This change allows the `VOLUME` and `SURFACE` kernel 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 shared and non-shared data and accumulates the values using Runge-Kutta constants to produce the field values for the timestep. The sequence of operations showing the overlapped execution of the `SURFACE` and `VOLUME` kernel is shown in Figure 2.7.

Another improvement introduced before the thesis 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 aliases `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 whereas `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 2.3, exchanging the functionality of the buffers for the next iteration.

This concept helps to improve the performance of the `RK` kernel by removing read and write memory dependency on the same buffer and also allows executing `RK` individually for shared

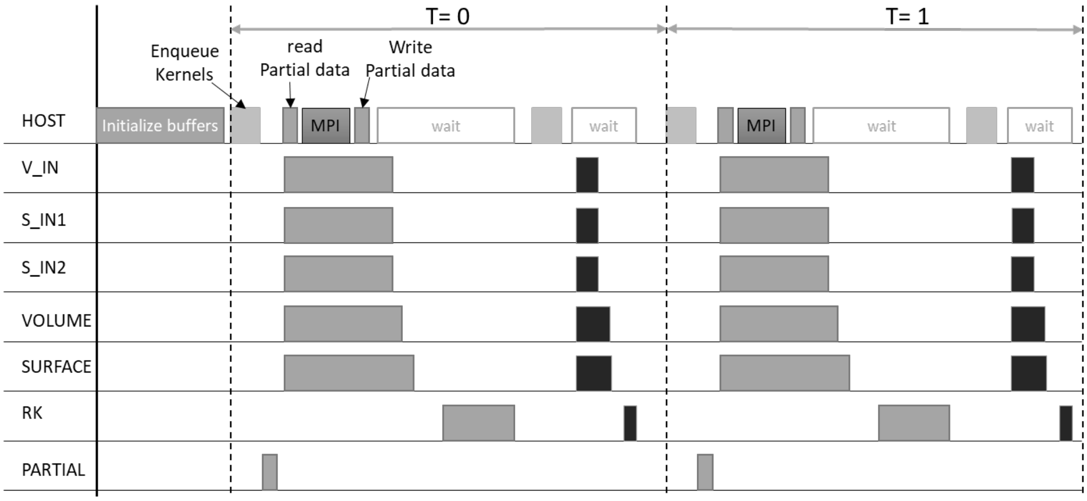


Figure 2.7: Sequence of `MIDG2 MPI FPGA` OpenCL kernels after including `volrhsQ` and `surrhsQ` showing them executing simultaneously

Listing 2.3: Buffer switching for double buffers in each iteration

```
for (itr = 0; itr < MaxTimeStep; itr++)
{
    if (buffSwitch)
    {
        set_argument("g_Q_in", Q1_pong_mem);
        set_argument("g_Q_out", Q1_ping_mem);
    }
    else
    {
        set_argument("g_Q_in", Q1_ping_mem);
        set_argument("g_Q_out", Q1_pong_mem);
    }
    buffSwitch = !buffSwitch;
}
```

and non-shared data as shown in Figure 2.7. 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. After computation is finished, `RK` kernel 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 `MIDG2 MPI FPGA` is shown in Figure 2.8.

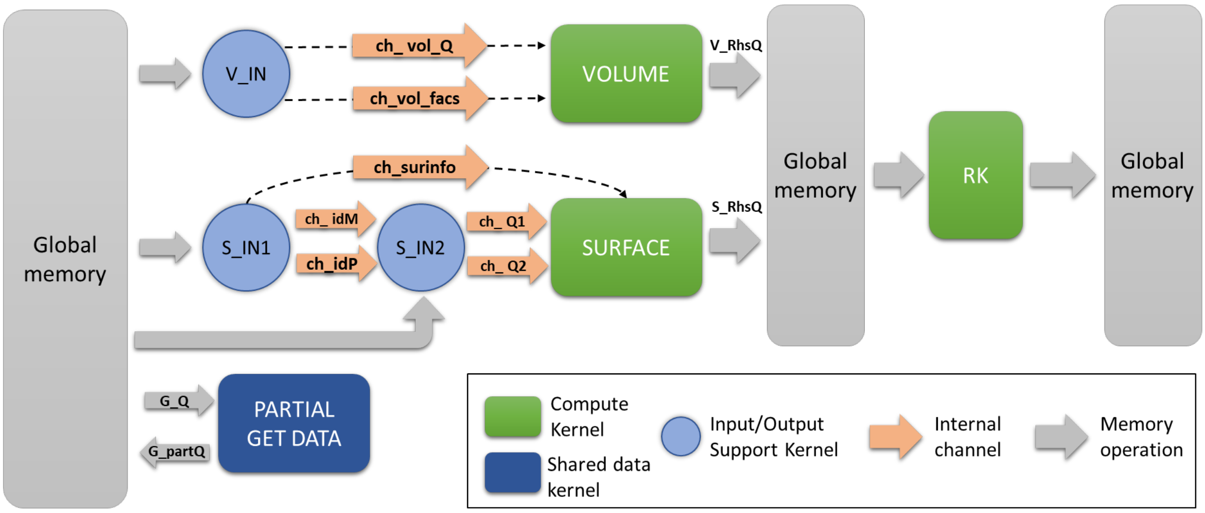


Figure 2.8: Structure for `MIDG2 MPI FPGA` OpenC kernels showing the new partial get kernel used to gather the shared elements into a separate buffer `g_partQ` for host to read. It also shows the introduction of `volrhsQ` and `surrhsQ` which allows the kernels to execute simultaneously as shown

Chapter 3

Topologies for FPGA-to-FPGA communication

This chapter describes 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. The last section presents the evaluation setup and the results of the evaluation of the bandwidth for the implemented topologies.

3.1 Topologies

As the current available BSP for the BittWare 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 application` 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.

Terms used

To make the understanding of the topologies clear, this section introduces 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 the FPGA topology 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 `FPGA` 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. This topology can be scaled by adding more isles which communicate

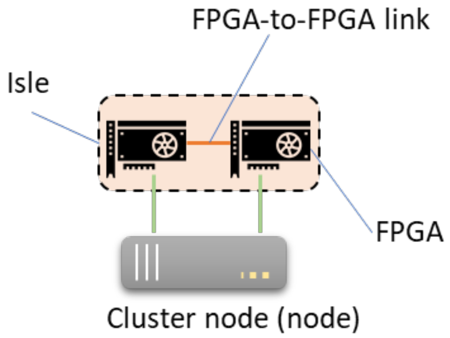


Figure 3.1: Simple network showing the network components

to each other using MPI via the `node` using Intel Omni path network. The topology is simple and easy 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.

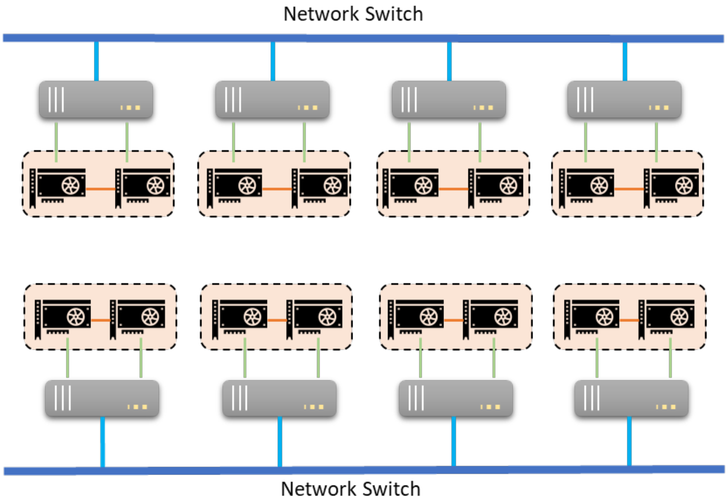


Figure 3.2: Within Node topology with two node per isle

3.1.2 Fully connected

The second topology extends the single `node` topology to two `nodes` such that an isle contains four `FPGAs fully connected` to each other with separate 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+PCIe, the data should be collected on a single `FPGA` using the point-to-point link and then exchanged via MPI+PCIe. On the receiving end the process can be reversed.

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.

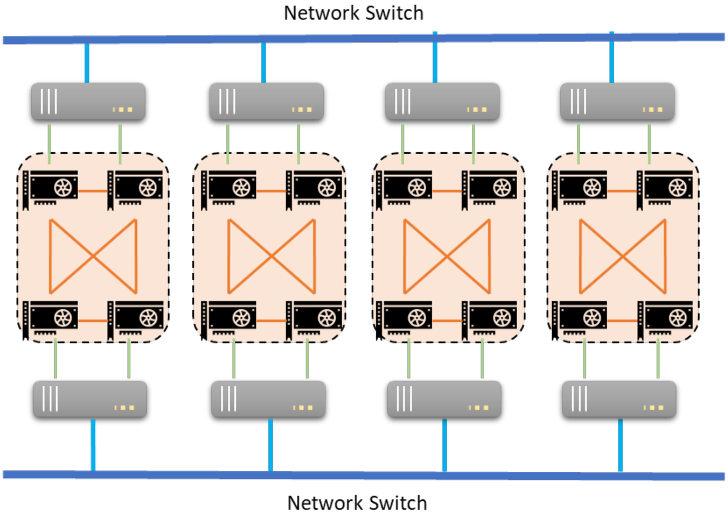


Figure 3.3: `Fully Connected` topology of four FPGAs per isle

3.1.3 Connected Graph

This topology is an extension 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 shown in Figure 3.4. In this topology all the `FPGAs` can communicate to each other without requiring any data communication via `host application`. 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. The 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.

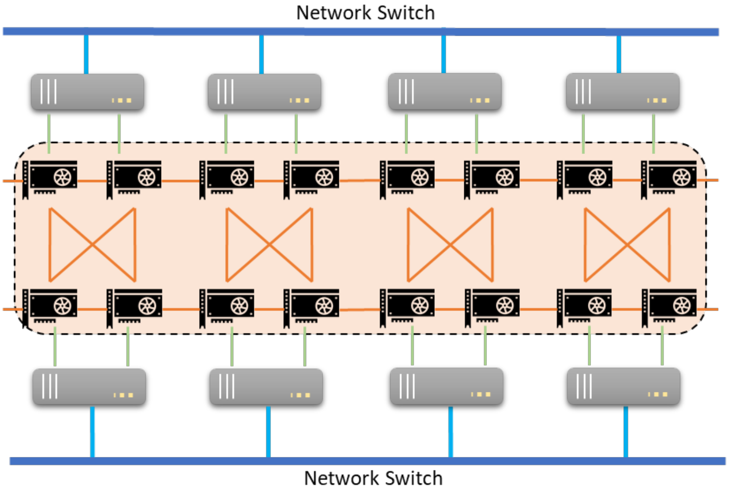


Figure 3.4: Connected Graph with HOPs between `fully connected` groups

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 [22], 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 connections 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 hop in each direction. The actual routing and 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.

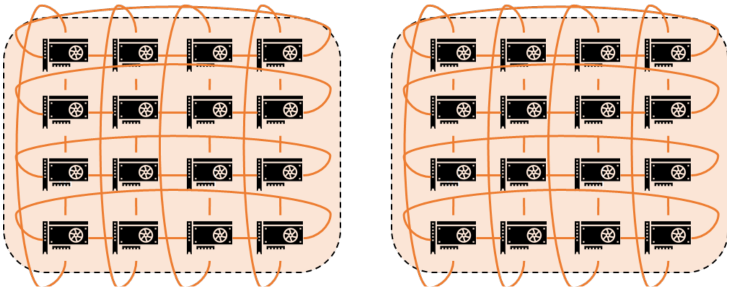


Figure 3.5: Two Toroidal network to connect 32 FPGAs

3.2 Prototypes to evaluate topologies

This section describes 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 application` code through the parameters `input` and `length` respectively. The `host application` 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 application` using `enqueueReadBuffer()` to complete the data exchange between the FPGAs.

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.

Listing 3.1: Kernels for `within node` prototype

```
#pragma OPENCL EXTENSION cl_intel_channels : enable

channel float8 ch_eth_in __attribute((io("kernel_input_ch0")));
channel float8 ch_eth_out __attribute((io("kernel_output_ch0")));

__kernel void __attribute__ ((max_global_work_dim(0)))
sender(int length, __global float8 * restrict input)
{
    for(int i=0; i<length; ++i)
        write_channel_intel(ch_eth_out, input[i]);
}

__kernel void __attribute__ ((max_global_work_dim(0)))
collector(int length, __global float8 * restrict output)
{
    for(int i=0; i<length; ++i)
        output[i] = read_channel_intel(ch_eth_in);
}
```

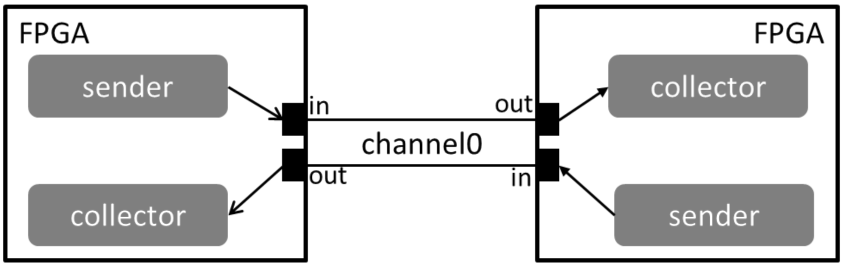


Figure 3.6: Communication structure of the kernels for `within node` topology with one channel

The `host application` was implemented using OpenCL C++ APIs 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 application` 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 multiple parallel transfers of the data 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`. The `OpenCL KERNEL` for the `fully connected` topology uses additional 2 channels for communicating with two other FPGAs. The prototype uses fixed channel mapping 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 the `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 a different `FPGA` simultaneously. For the prototypes, the `host application` is designed to communicate the same amount of data to each `FPGA` although the kernels support sending and receiving different data sizes from each channel. The `host application` was limited to use the 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 mesh partition information created in the `host application` which will allow configuring the size dynamically.

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`

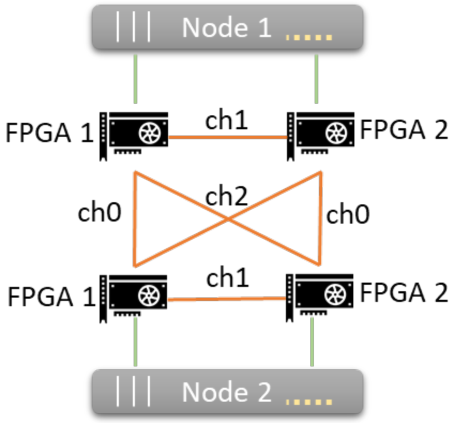


Figure 3.7: Hardware setup for the `fully connected` topology

Listing 3.3: Channels used for `fully connected` topology

```
#pragma OPENCL EXTENSION cl_intel_channels : enable
channel float8 ch_eth_in0 __attribute((io("kernel_input_ch0")));
channel float8 ch_eth_out0 __attribute((io("kernel_output_ch0")));

channel float8 ch_eth_in1 __attribute((io("kernel_input_ch1")));
channel float8 ch_eth_out1 __attribute((io("kernel_output_ch1")));

channel float8 ch_eth_in2 __attribute((io("kernel_input_ch2")));
channel float8 ch_eth_out2 __attribute((io("kernel_output_ch2")));
```

`application`. The kernels iterate over the input buffers for the maximum length among the three lengths, sending 256 bits in each iteration over each channel. Individual length for each channel is used to stop the sends for the specific channel in following iterations. The structure shown in the Listing creates write units for three channels and three memory load Load store unit (LSU)s which allows sending the data parallely on all the three channels in each iteration.

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.

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

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 application` starts the kernel using the `cl::CommandQueue::enqueueNDRangeKernel()` method. The `host application` waits for the completion of collector kernels on each FPGA, and then reads the received data for verification. Additionally `MPI_Barrier(MPI_COMM_WORLD)` function is used the synchronize the `host applications` before executing the kernels of the FPGA device. The `host application` before enqueuing the kernels for communicating the data waits for the other `host application` to finish the initialization or processing from the previous iteration. As the external channels are blocking in nature, if the kernels communicating

Listing 3.4: Sender Kernel for `fully connected`

```
__kernel void __attribute__ ((max_global_work_dim(0)))
sender_all(int length, int length1, int length2,
            __global float8 * restrict input,
            __global float8 * restrict input1,
            __global float8 * restrict input2)
{
    for(int i=0; i<length || i < length1 || i < length2; i++)
    {
        if (i < length)
            write_channel_intel(ch_eth_out0, input[i]);

        if (i < length1)
            write_channel_intel(ch_eth_out1, input1[i]);

        if (i < length2)
            write_channel_intel(ch_eth_out2, input2[i]);
    }
}
```

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);
    }
}
```

with the external channel are not synchronized, one of the kernels will perceive higher stalls. This additional synchronization helps to reduce the stalls on the external channels as the kernels on all the FPGAs start approximately at same time reducing the waits.

3.3 Evaluation of the Topologies

The first set of evaluation tests was carried out using the prototypes developed for the different topologies which are described in this chapter. The aim of the tests was to measure the bandwidth possible to achieve with the developed OpenCL kernels for the topologies and compare it with the communication architecture which utilizes `HOST` to communicate the FPGA data.

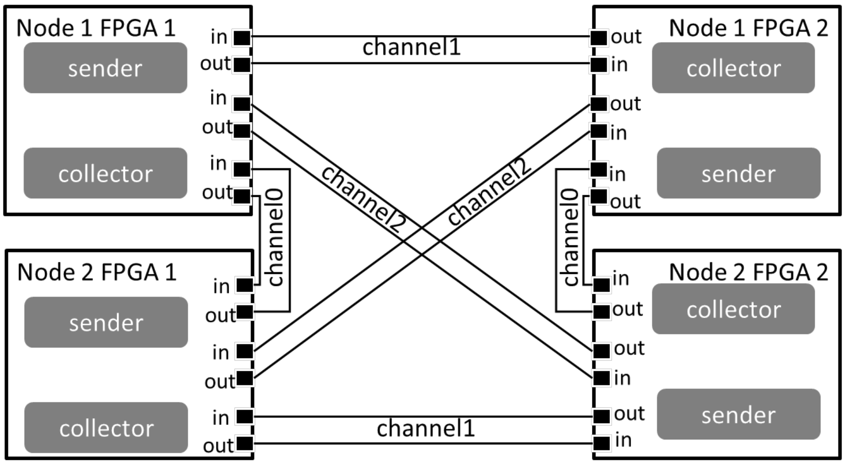


Figure 3.8: Communication structure for the `fully connected` kernels on each FPGA

3.3.1 Setup

The experiments were conducted with the FPGA partition of the Noctua cluster. 4 Nodes at maximum was utilized. Each of the nodes contain 2 Intel Xeon Gold "Skylake" 6148(F), 2.4 GHz CPUs and 2 Bittware 520N[1] cards which have Intel Stratix 10 FPGA with 32 GiB memory. The Nodes are connected with Intel Omni Path 100 Gbps network which is used for data communication between the CPUs using MPI. The FPGAs are connected using a point-to-point optical transceiver and optical fiber links in the specified topologies.

The bandwidth evaluation is done by communicating data of different sizes between the nodes in a ping-ping pattern in all the designs. The ping-ping pattern is the ideal candidate as it maximizes the utilization of the duplex point-to-point channels in the FPGAs. To use as a reference to the current MIDG2 implementation, an additional prototype is developed which performs the communication between the FPGAs via HOST. In this prototype, the data from the FPGAs is read over the PCIe and transferred by the HOST over the Intel Omni path network to the other HOST. On receipt, the second HOST copies the data into the FPGA over the PCIe to complete a single data transaction. The data communication between HOSTs is performed using Intel MPI Library v18.0.3. Listing 3.6 shows the pseudo-code of the section performing the communication. This prototype simulates the data communication behavior in the base MPI MIDG2 FPGA implementation and is used to compare the bandwidth performance and possible improvements using FPGA-to-FPGA communication on the target hardware systems. The evaluation performed in this section compares the bandwidth of the FPGA-to-FPGA communication in different topologies with the Intel® Omni Path + PCIe based data communication between the nodes.

Two different types of data sizes are used for the evaluation. The first set of data sizes named as `regular` are multiples of 32 which produce a aligned data transactions on the FPGA channels without requirement of any extra padding or alignment. The second set named as `irregular` are data sizes taken from the actual MIDG2 communication pattern for 2 node system. These data sizes produce non-aligned communication on the channels and are explicitly required to be aligned/padded to ensure aligned 32 byte writes and read on the channel. The use of the different data size types was to evaluate the effects of additional padding requirements which is necessary in most of the applications.

[1] https://www.bittware.com/fpga/520n/

Listing 3.6: Pseudo-code to perform MPI+PCIe based data communication between FPGAs

```
clock_gettime(CLOCK_MONOTONIC, &execStart);
readQ.enqueueReadBuffer(readBuffer, CL_TRUE, 0, buffersize*(nprocs), send_buffer);
for (int p: nodeList)
{
    if (p != procid)
    {
        MPI_Isend(send_buffer[p], buffersize,, p,,);
        MPI_Irecv((rcv_buffer[p], buffersize,, p,,);
    }
}

MPI_Waitall(req, mpi_in_requests,);
MPI_Waitall(req, mpi_out_requests,);

processBuffQ.enqueueWriteBuffer(writeBuffer, CL_TRUE, 0, buffersize*(nprocs), rcv_buffer);
clock_gettime(CLOCK_MONOTONIC, &execEnd);
```

Another set of variations evaluated with the FPGA prototypes is the use of `interleaved` or `non-interleaved` memory partitions for global memory of the FPGA. The global memory on the Bittware 520N board contains 4 channels to access global memory. The channels can be either configured to be used in a bus-interleaved or non-interleaved fashion. The non-interleaved global memory access allows the user to specify the memory channel to be used for accessing the specific buffer. These memory partition affects the global memory access bandwidth and the topologies were compared to identify the configuration which performs best for the topology. The evaluation was done for 5 designs with each having 4 variation giving a total of 20 different data sets listed with description for understanding the following data

3.3.2 Bandwidth Comparison

The bandwidth in the topology designs is computed by measuring the execution time of the `send` kernels using `CL_PROFILING_COMMAND_START` and `CL_PROFILING_COMMAND_END` profile counters which give the start and end time of the kernels in nanoseconds. The ping-ping pattern is used for the communication in which all the nodes start sending data simultaneously. As FPGAs have full duplex channels for communication, there would be no interference observed for the send and receives. For each data size the communication is performed for 100 rounds. The buffer to be exchanged is initialized every round with a value derived from the element index and the index of the round in which the data is transferred. On receipt the data is checked to verify the that the data is transferred correctly over the channels or the Intel Omni Path network + PCIe. After each execution, the execution time of the send kernel is computed using the counters values and the bandwidth for that run is computed using the formula

$$bandwidth(MB/Secs) = \frac{DataSize(inMB)}{exectime(inseconds)}$$

The bandwidth is accumulated for each round and then the average bandwidth for the data size is computed after the end of the 100 rounds and stored in file.

The computation in the MPI+PCIe prototype is similar but instead of using the kernel execution time, the complete time for data transfer from FPGA->CPU over PCIe, MPI transfer and data transfer from CPU->FPGA over PCIe is used. This time is computed using `clock_gettime` as shown in the code 3.6.

The theoretical peak values for each of the communication patterns is listed in the Table 3.1. The peak bandwidth of the MPI+PCIe design is a combination of the bandwidths of the PCIe and the Intel® Omni Path. As the data is transferred between the FPGA to HOST and HOST to FPGA in a store and forward manner, the effective bandwidth of the communication can be computed as explained in [19] and is given by the formula:

$$P_{mpipcie} = \frac{N}{\frac{N}{P_{PCIe}} + \frac{N}{P_{IOP}} + \frac{N}{P_{PCIe}}} = \frac{N}{\frac{N}{7.88GB/s} + \frac{N}{12.5GB/s} + \frac{N}{7.88GB/s}} = 2.995GB/s \qquad (3.1)$$

Each external channel of the FPGA is operated at 40 Gbits/s giving a total of 20 GB/s peak bandwidth for all the channels. Within node topology either uses 1 channel or all 4 and in the `fully connected` topology only three channels is used.

Table 3.1: Peak bandwidth in each configuration

Configuration	**Peak Bandwidth (GB/s)**
MPI+PCIe	2.995
Within Node 1 Channel	5
Within Node 4 Channel	20
Fully Connect	15

Plots in Figure 3.9 shows the bandwidth variation for Within node design using single channel and all the 4 channels. Irregular data sizes perform similar to regular data sizes which suggests that there are no major overhead involved with the handling of the irregular data types in applications. For the single channel design, there are no effects of the global memory interleaving as the slower external channel is the bottleneck instead of the global memory. On the other hand, for the 4 channels, use of non-interleaved memory affects the bandwidth performance and the effective bandwidth peak bandwidth is 16.41 GB/s which is 82 percentage of the peak bandwidth. This is due to the overheads and lower occupancy of the global memory reads/write from the send and write. The kernels read/write 4*256 bits per request from the same memory channel

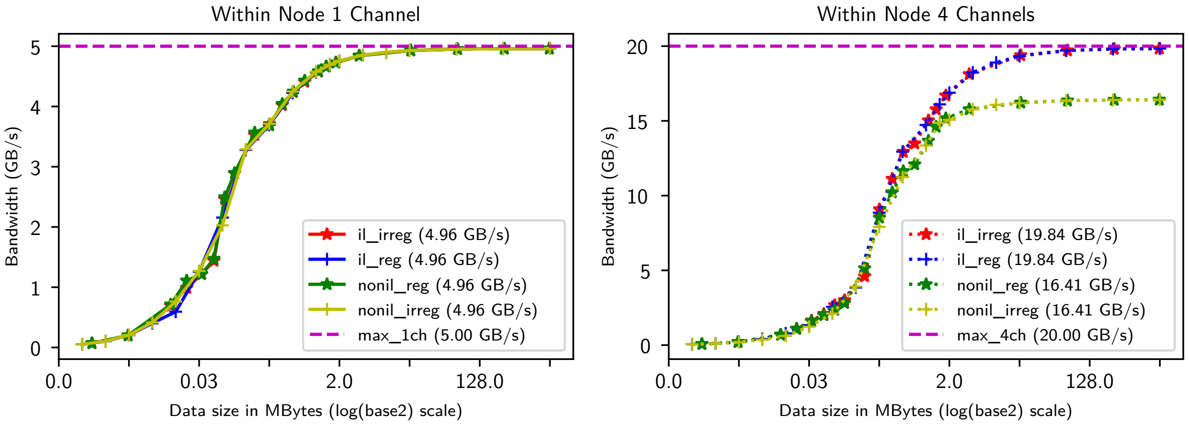


Figure 3.9: Bandwidth variation of `Within Node` designs. The `il_xxx` in the name is used for variants with interleaved memory and `nonil_xxx` for non-interleaved memory. `xxx_reg` variant uses regular data sizes for the tests and `xxx_irreg` uses Irregular data sizes

which is not possible. The global memory channels have a maximum width of 512 bits and requests of memory width wider sizes would increase the latency and stalls. While using the interleaved memory, the buffers are stored in all the four memory channels and the kernels can access them simultaneously. This reduces the latency and stalls for the reads and writes allowing to reach the active bandwidth of 19.84 GB/s which is 99.2 percentage of the peak bandwidth.

For the `fully connected` topology also, the bandwidth is not affected by `irregular` data sizes. The `non-interleaved` and `interleaved` memory designs have similar bandwidth performance for each data type This is because the memory channels are explicitly mapped to use separate memory channel for each buffer in the host application. The explicit mapping assigns individual memory channels to each buffer in global memory. Each of the memory is used only to read/write data to/from a specific external channel which ensures parallel reads and writes without latency and stalls. Separate memory channels were used to show the full potential of the `fully topology` but another set of tests were performed by using a single memory channel for all the buffers. This variant evaluates the bandwidth performance of the fully connected topology in a typical application use case. In an application such as the MIDG2, the global memory has multiple buffers used by many kernels and separating the data for each external channel into individual buffers mapped to separate memory channel could not be trivial.

The peak bandwidth achieved with the variants using multiple channels or interleaved memory of `fully connected` design is 14.88 GB/s which is 99.2 percentage of peak 15 GB/s bandwidth possible with 3 channels. The peak bandwidth achieved by the single memory channel variant is only 7.27 GB/s which less than half of the peak bandwidth. The plot in the Figure 3.10 shows the variation of the bandwidth over the data sizes. The error bars are added to plot using

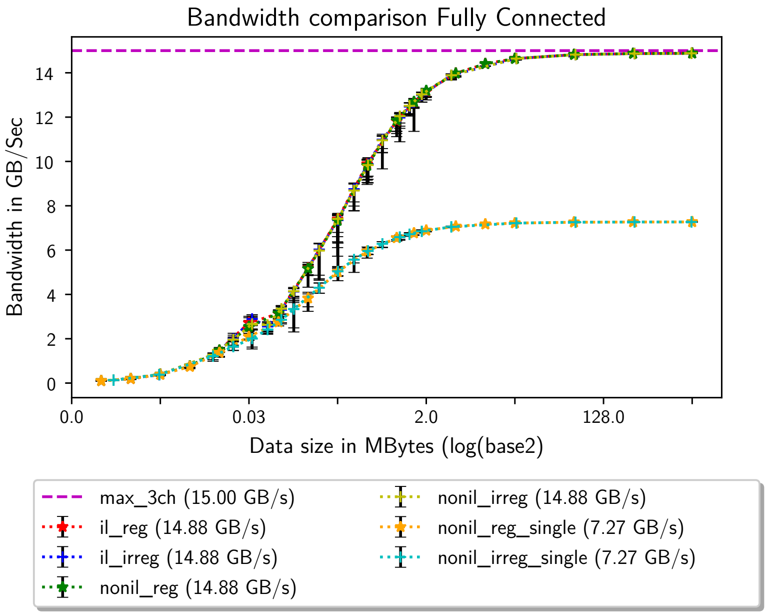


Figure 3.10: Bandwidth variation of Fully connected design. The `il_xxx` in the name is used for variants with interleaved memory and `nonil_xxx` for non-interleaved memory. `xxx_reg` variant uses regular data sizes for the tests and `xxx_irreg` uses Irregular data sizes. The `xxx_xxx_single` variant uses a single global memory channel for the buffers linked with all the three external channels similar to MIDG2 application

the measured minimum and maximum bandwidths to show the actual variations. The bandwidth has higher variations for data sizes below 2 MB with a maximum of 2.76 GB/s difference for 256 Kbytes size. We can see that the fully connected topology can achieve high bandwidth with almost 100% efficiency if the global memory is available for parallel reads and writes. The efficiency decreases to 50% when only one memory is available for all the channels as the global memory cannot handle multiple parallel requests on the same channel due to congestion and stalls the IO operations.

The MPI prototypes are only able to achieve a maximum peak bandwidth of 1.94 GB/s with 2 Node system which is only 65% of the peak bandwidth. The 4 node system only achieves a peak bandwidth of 1.78 GB/s. The main reason for the slower overall bandwidth with 4 nodes is due to the larger data transfers between the host and the FPGA. The latency of the communication between FPGA-HOST and HOST-FPGA is much larger than the communication between two HOSTs. There is also a huge difference in the minimum and maximum bandwidths for all data sizes as it can be seen from the longer error bars in the Figure 3.11.

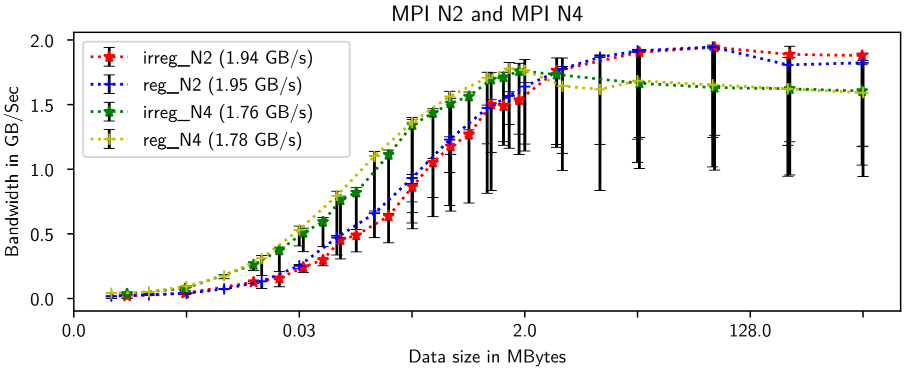


Figure 3.11: Bandwidth variation of MPI+PCIe design, reg_xxx variant uses regular data sizes for the tests and irreg_xxx uses Irregular data sizes. xxx_N2 refer to system using two MPI rings and xxx_N4 for 4 MPI rings

The peak bandwidth observed for the designs is summarized in table 3.2 and a plot comparing the variation of the bandwidth over data size as shown in Figure 3.12. The IO channels prototypes can utilize the 99.2% of the peak bandwidth of using regular and non-regular data sizes in ping-ping communication pattern. The MPI+PCIe based communication is only able to utilize the 65% of the available bandwidth which shows that this communication is less efficient and can be clear bottleneck for applications which require large data communications.

Designs	Regular		Ir-regular	
	Interleaved	Non-interleaved	Interleaved	Non-interleaved
WN 1CH	4.96	4.96	4.96	4.96
WN 4CH	19.84	16.41	19.84	16.41
FC	14.88	14.88	14.88	14.88
MPI N2	1.95		1.94	
MPI N4	1.78		1.76	

Table 3.2: Peak bandwidth observed for each design variant

The plot in the Figure 3.12 also highlights the bandwidth values of each design at 512 KB data transfer to compare the performance for smaller data sizes which is typical for some applications.

Within node design with 1 channel achieves a bandwidth of 4.22 GB/s which is 84% of the peak bandwidth whereas other channel design though utilizing more number of channels is only able to achieve 62%(FC) and 64%(WN 4CH). When compared with `fully connected` topology, the main reason for better efficiency is identified as the effect of stalls on the channels. As the kernels in the `within node` topology is controlled by single host application, the communication on the channels is synchronized and produce lesser stalls. For `fully connected` topology, the FPGAs is controlled by 2 separate `Host applications` which execute independent to each other. This leads to un-synchronized read and write on the IO channels as the host controls when the kernels execute and access the channels. This tends to increase the number of stalls as the channels are blocking in nature leading to decrease in efficiency of the communication. The lower percentage for `within node` topology is due to the design of the kernels. In the all the tests, same amount of data is transferred over 4 channels as transferred over one which makes the efficiency lesser as per channel smaller data sizes are communicated. This is done to keep the communication pattern similar between the nodes in all designs and scaling happens only by adding more nodes instead of communicating larger data per transfer.

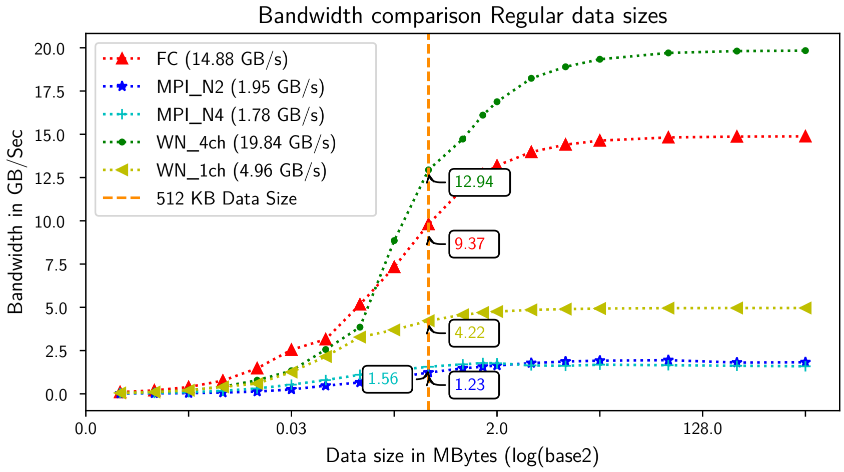


Figure 3.12: Comparison of bandwidths of all the designs communicating with regular data sizes. The texts show the bandwidths achieved for 512 KBytes data transfers

The IO channels are capable of very high bandwidth when compared to the MPI+PCIe communication. A single IO channel itself has 66.6% higher overall bandwidth then the MPI+PCIe communication. Looking at the efficiency of the communication achieved in the evaluation, the `within node` topology which uses single channel for communication would result in 3 times faster communication and with 4 channel 10 times faster communication then MPI+PCIe. This is because the IO channels are efficient and can be utilized to use almost complete bandwidth (5 GB/s and 20 GB/s) available for large data transfers compared to only 65% efficiency of MPI+PCIe (2.99 GB/s). The `fully connected` topology also has 5 times bigger overall bandwidth then MPI+PCIe for communicating with 3 nodes simultaneously. The efficiency of the MPI+PCIe reduces to 60% with 4 nodes whereas with `fully connected` topology communication between 4 nodes is performed with no changes in performance of the channels. Also, the communication in `fully connected` topology occur parallelly between the nodes compared to routed communication via a network switch which adds latency to the communication. Bandwidth evaluation using the prototypes implemented in this chapter show the potential benefits of using the IO channels instead of MPI+PCIe for communication between FPGAs to reduce the communication time and increase efficiency.

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 2. The thesis extends the MIDG2 MPI FPGA version of the application introduced in the section 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 but uses the MPI+PCIe to communicate which have low bandwidth as shown in section 3.3. To improve the bandwidth performance of the application, the IO channels are proposed to be used. This chapter explains the changes which are 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 to create a `MIDG2 FPGA IO channels` design and evaluate the benefits of the direct communication with the MIDG2 application.

4.1 Kernel Structure with IO Channels

The first set of changes done in the thesis to the `MIDG2 MPI FPGA` application was to include the IO channels to exchange the shared data in a multiple FPGA system. The IO channels are used to exchange the shared data between the FPGAs directly instead of copying it first to global memory, which is read by the `Host application` and then transferred using MPI. The prototypes developed for the topology's evaluation serve 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 it 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` and `fully connected` are developed. The `within node` scales the application to two FPGAs and the `fully connected` to 4 FPGAs.

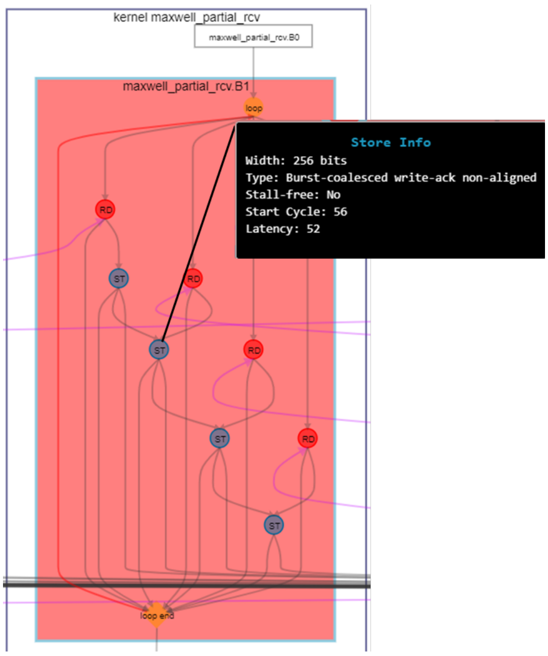
As explained in the section 2.5.1, `partial_get_kernel` reads the shared from the `g_Q` memory and writes them back into another buffer to coalesce the shared data. The new `partial_send` and `partial_recv` split these operations and also perform the communication with the other FPGA over the IO channels. The `partial_send` kernel reads the shared elements from the `g_Q` buffer and sends the shared elements data over a single or multiple external channel to other

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

FPGA(s). The partial_recv receives the data from other FPGAs over the channels and then stores it in g_partQ buffer which is passed to the SURFACE kernel for processing.

In the within node topology, the data is only exchanged between two nodes which simplifies the channel assignment for the shared data. All the shared data is sent and received to/from only one node using one or four channels. In the fully connected topology, the communication is not straight forward. In a 4 node system, the shared elements are mapped to different nodes. The partial_send and partial_recv kernels need this mapping information at the runtime to transfer and receive data correctly to/from target node. The mapping information for identifying the indexes to be read from the g_Q is provided in another buffer g_index. g_index contains the list of the indexes of the shared data sorted by the target MPI rank. g_index was used in the partial_get_kernel of the MIDG2 MPI FPGA application to identify the shared data. For IO channels design along with g_index, additional information for mapping the indexes to specific target FPGA is passed by using two parameters start<x> and nout<x> for each channel x. In the partial_send kernel, the start<x> specifies the start location within g_index and the nout<x> specifies the number of indexes to send for the channel x. In partial_recv the start<x> specifies the start index within the g_partQ and the nout<x> specifies number of index to receive for channel x. As each channel is assigned to a specific FPGA node, start<x> and nout<x> map the data for a target FPGA to the channel used to connect with the target FPGA.

With this information, The partial_send is able to read data from g_Q for all channels simultaneously, and send them over the respective channel. Also, the partial_recv kernel is able to receive the data and write them into g_partQ simultaneously. The first version of the partial_send and partial_recv kernels used this information for mapping the memories to channels but memory dependencies were reported in partial_recv kernel by the Intel OpenCL FPGA compiler. As the received data in the partial_recv is written into the same g_partQ buffer, the Intel OpenCL compiler identified the multiple memory writes as interdependent and serialized the channel read and memory writes as shown in Figure 4.1. As the data from each



Store Info
Width: 256 bits
Type: Burst-coalesced write-ack non-aligned
Stall-free: No
Start Cycle: 56
Latency: 52

Figure 4.1: Memory dependency resulting in serial channel reads in the partial_recv for fully connected design

channel is written into different location in the buffer (different `start<x>` values), there is no dependencies or overlaps the parallel writes would not create any memory corruption. The Intel OpenCL compiler was unable to understand it from the existing structure and also there is no way to correctly specify this in the kernel code. So, to resolve this issue, `g_partQ` buffer was aliased into four different names `g_partQ[1-4]`. Each of the `g_partQ<x>` buffer is used for individual writes of the data received from channel `x` though they are initialized by the same OpenCL buffer in the `Host application`. This change allowed Intel OpenCL compiler to create a sequence with parallel writes to the `g_partQ` buffer.

The addition of double buffer in MIDG2 MPI FPGA allowed improving the execution of the kernels by reducing the load on the `g_Q` buffer and executing `RK` kernel immediately after completion of computation for both shared and non-shared data separately. Though there was improvement, `RK` still executed after the compute kernels which was not desirable. Further optimization by introducing Intel OpenCL channels between `VOLUME/SURFACE` to `RK` to replace `volrhsQ` and `surrhsQ` was possible but was not implemented before the start of the thesis. So, this optimization to the kernel pipeline is introduced in this thesis as part of the IO channel implementation which could help to improve the performance of the kernels marginally. The OpenCL kernels implementation is updated to include internal channels to communicate the right-hand side field values from `VOLUME/SURFACE` to `RK` as shown in the kernel structure in Figure 4.2. 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.

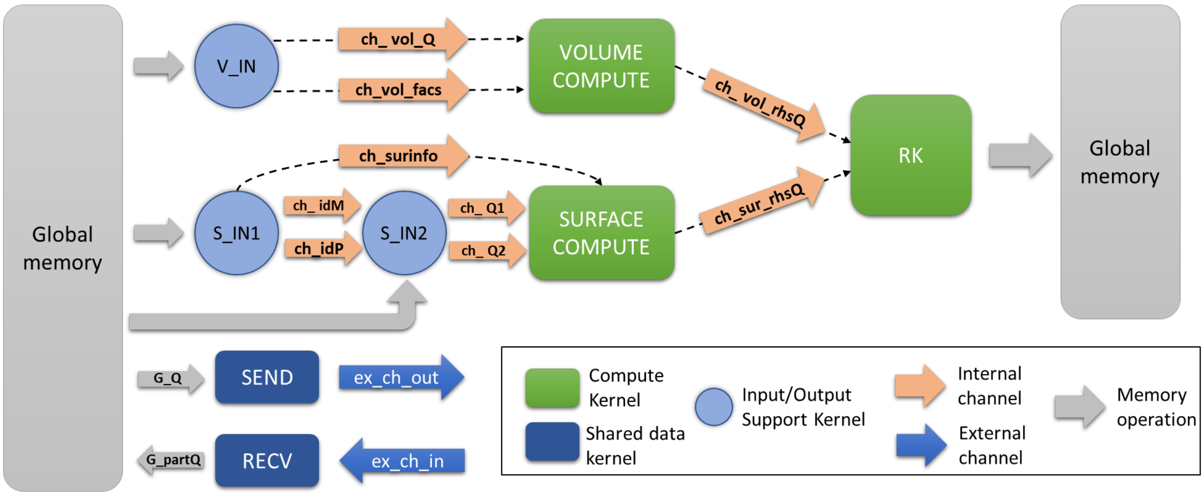
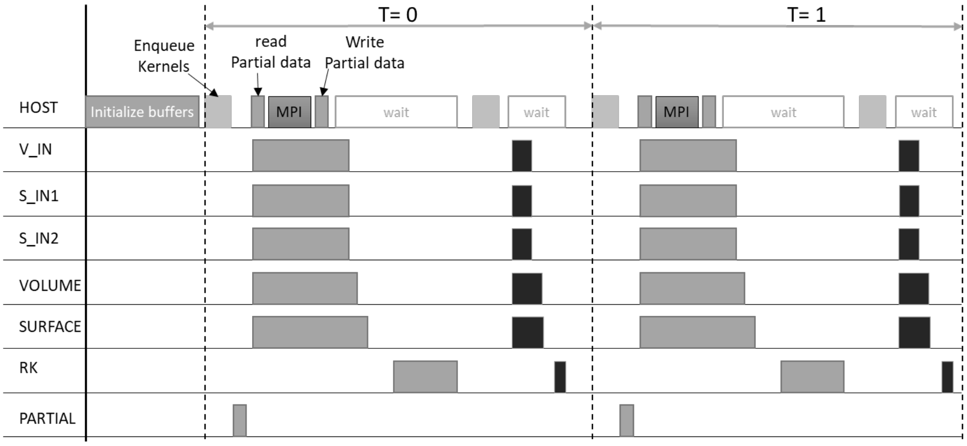
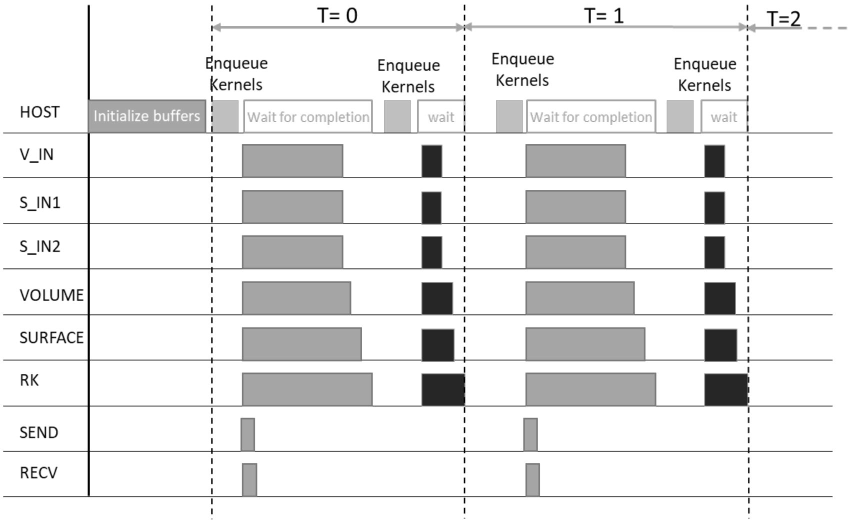


Figure 4.2: Structure for MPI FPGA OpenCL kernels showing `send` and `recv` kernels used for communication with external IO channels. Image also shows the internal channels introduced between `VOLUME/SURFACE` to `RK` kernel

In the IO channels design, the computation of shared elements follows the computation of non-shared elements as in the `MIDG2 MPI FPGA` design. 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 `MIDG2 MPI FPGA` and the IO channels design is shown in Figure 4.3. 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.



(a) MIDG2 MPI FPGA



(b) MIDG2 FPGA with IO channels

Figure 4.3: Sequence of Kernel and host event for MIDG2 MPI FPGA and MIDG2 FPGA with IO channels

4.2 Host changes to support IO Channels

The host application of MIDG2 application implements the 3D DG-method introduced in the section 2.2. 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 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 platform and device initialization code sections are updated to handle both the designs with the same host code. A hierarchical structure for the initialization code is implemented such that, the generic functionalities which include OpenCL platform identification, context creation etc is separated from the design specific OpenCL buffer initialization and kernel execution. The new structure of the host code is shown in Figure 5.5 and would be explained in detail in the section 5.2.2.

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. This 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 4.1.

Listing 4.1: 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 4.4 shows the logic of the external channel assignment at each node.

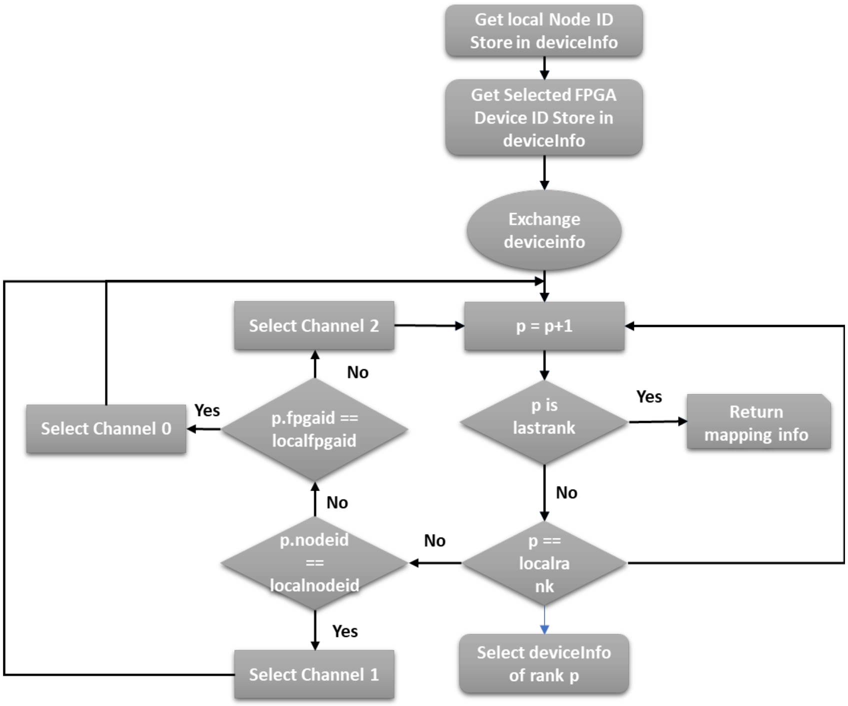


Figure 4.4: Channel selection logic for fully connected topology

Chapter 5

Removing host dependency for optimization

In all the previous versions MIDG2 FPGA, MIDG2 MPI FPGA and the first version with IO channels introduced in chapter 4, 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 to enqueue subgroup compute_inner and other to enqueue subgroup compute_halo shown in Listing 4.1. 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.

As the host was not required anymore for communication, further optimization of the kernel structure is done to remove the host application interaction completely. The change introduced is to implement the the timestepping loops in the kernels and create a structure which only requires the host application to interact twice per application invocation with the FPGA. Moving the timestepping loops to kernels additionally required to implement a synchronization construct between the kernels to be able to process the shared and non-shared data sequentially. This chapter introduces the modifications to kernels and host application to achieve this FPGA only design. The chapter first explains the kernel structures considered for implementing achieve a FPGA only system along with the issues identified and their fixes. The chapter then discusses the final FPGA only designs highlighting the optimization and changes from the previous versions in kernel and host code In the last it 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

The first change required to remove the host dependency in the FPGA kernels was to move the timestep loops to the kernels so that they can iterate for the specified timesteps without needing the host to start them every iteration. Implementing the timesteps in the kernels was easy. The timestep loops are added at the top level above the existing code as shown in the pseudo code in 5.1. To restrict the Intel OpenCL compiler from executing multiple iterations of the timestep iterations and Runge-Kutta stages, the #pragma max_concurrency 1 is added to the loops. This Intel OpenCL directive, limits the concurrency of the loops to 1. The complete modifications done to kernel codes is explained in section 5.2. The actual timesteps are still computed in the host application and are passed as parameter NSteps to kernels from the host. The actual value of the timestep is only known after the kernels are enqueue and loop optimization are not

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
        }
    }
}
```

possible. 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 presents the designs evaluated to identify the best possible design for implementing the synchronization with minimum overhead.

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 4.2, 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 previous versions `MIDG2 FPGA`, `MIDG2 MPI FPGA` and `MIDG2 FPGA IO channels`, the `host application` performs these synchronizations. The host first starts the kernels to process non-shared data along with communication kernels which execute parallelly. The hosts waits for kernels to finish the processing the non-shared elements using the `waitforcompletion()`

method and for the MPI to finish communication using `MPI_waitall()` API before starting kernels again to process shared elements. This sequence of operations is shown in Figure 4.3 (a).

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 an 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.

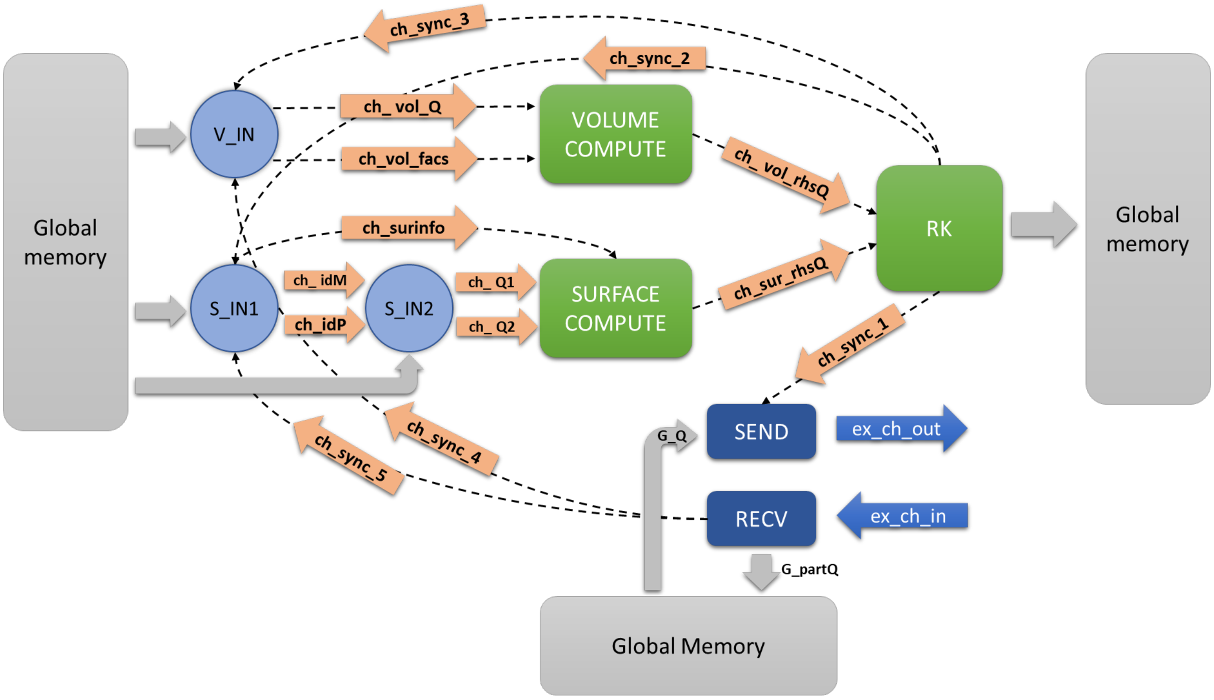


Figure 5.1: Kernel structure for FPGA only design utilizing 5 blocking channels for synchronization

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 supposed 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 parameters were marked as `volatile` which allows to remove cached access [**section 2.8.7**, 13] to ensure correct sequence of reads and writes. This allowed removal of the cache for all the read `LSU` but still deviation in the nodal errors were present and further analysis was done to identify the cause.

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 smaller than that of a memory operation. This would cause the input kernels to access non-updated memory after receiving the event over the channels. As shown in the Figure 5.2 (a), 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.

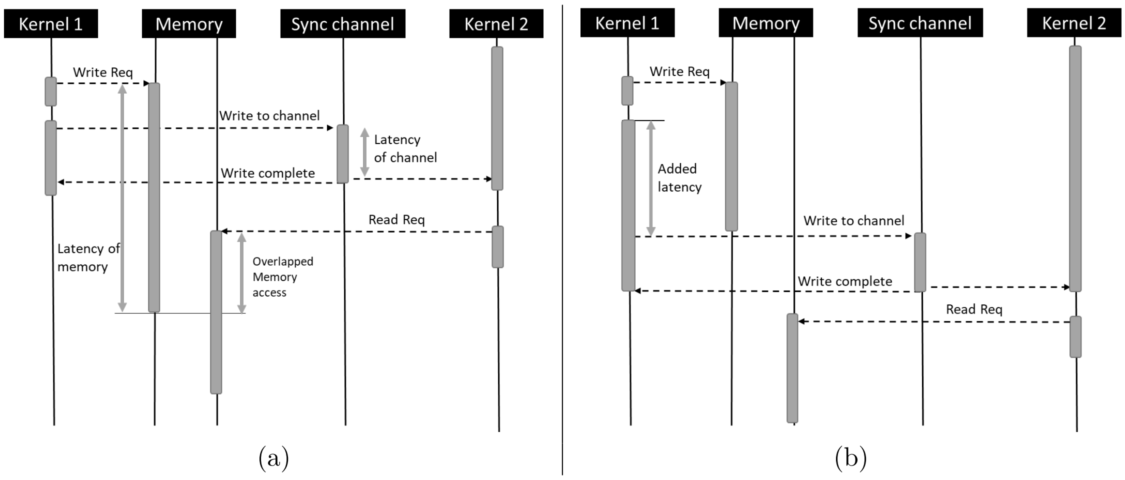


Figure 5.2: Sequence of operation between kernels to synchronize memory access with channels. (a) Shows the latency differences for memory and channel causing overlapped memory access. (b) Shows the sequence with added latency which avoids the overlap

As there is no concrete information available for the target board to estimate the exact latency for the memory and no 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

```
for(int time = 0; time < waitCount; ++time)
{
    if (time == waitCount - 1)
    {
        write_channel_intel(channel, token);
    }
}
```

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

```
int atomic_add(volatile __global int *p, int val)
int atomic_sub(volatile __global int *p, int val)
int atomic_xchg(volatile __global int *p, int val)
int atomic_inc(volatile __global int *p)
int atomic_dec(volatile __global int *p)
int atomic_cmpxchg(volatile __global int *p, int cmp, int val)
int atomic_min(volatile __global int *p, int val)
int atomic_max(volatile __global int *p, int val)
int atomic_and(volatile __global int *p, int val)
int atomic_or(volatile __global int *p, int val)
int atomic_xor(volatile __global int *p, int val)
```

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 5.2 (b) which shows no overlapping in the memory requests due to latency.

Removing channels ch_sync_2 and ch_sync_3

Another issue reported by the Intel OpenCL compiler as warning was the 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 after addition of the volatile keyword for the buffers as explained in 5.2.1, an implicit synchronization of the memory operations is created between kernels 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 a locking/unlocking structure for synchronization. Intel OpenCL FPGA SDK supports the standard OpenCL *Atomic Functions for 32-bit Integers* listed in 5.3.

Listing 5.4: Synchronization Implementation with atomic functions

```
#pragma max_concurrency 1
for (int step = 0; step < timesteps; ++step)
{
    #pragma max_concurrency 1
    for(int intrk = 0; intrk < 5; ++intrk)
    {
        // Wait for memory 1 to be cleared
        while (atomic_cmpxchg(&lock[1], 0, 0xFB) == 0xFB);

        // Clear memory 3
        atomic_xchg(&lock[3], 0);
    }
}
```

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 5.4. 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.

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 time 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 5.3.

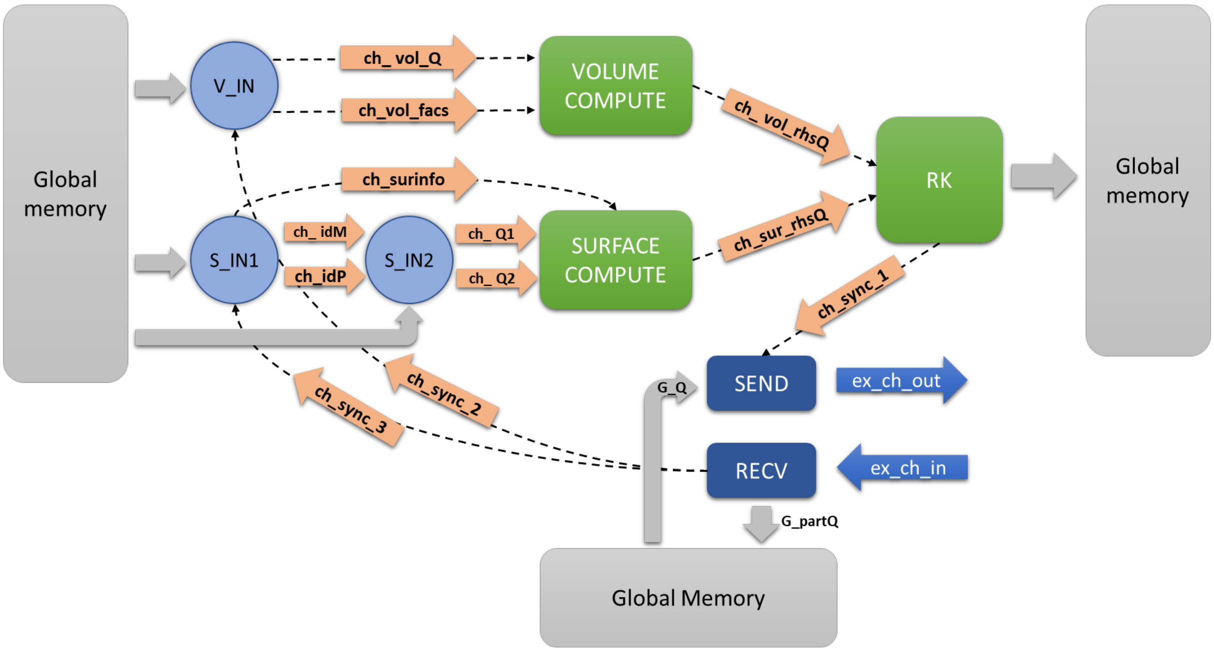


Figure 5.3: Kernel structure of the optimized FPGA only design

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 kth 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 the 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 stages 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 2.5 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 increase in the resource utilization as 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 kth 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 5.3. 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.

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

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

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

```
__kernel void rkkernel(..Args..)
{
    const float rk4a[5] = {Vs1,Vs2,Vs3,Vs4,Vs5};
    const float rk4b[5] = {Vs1,Vs2,Vs3,Vs4,Vs5};
    // Outer timestep loop
    #pragma loop_coalesce 2
    for (int step = 0; step < timesteps; ++step)
    {
        // 5 RK steps
        for(int intrk = 0; intrk < 5; ++intrk)
        {
            // Select the RK−coefficient for  the  stage
            float l_fa = rk4a[intrk];
            float l_fb = rk4b[intrk];

            // Old kernel code inside here
            // Process/Read/Write element
        }
    }
}
```

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.

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 line number 17 and 18 in Listing 5.7. The switching is done to switch the input buffer and the output buffer after every iteration which allows parallel execution of the all the kernels for computation.

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 (line numbers 23-27) 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 application` the same OpenCL memory buffer is assigned to both the parameters

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

```
__kernel void kernelName(__private int K,  __private int arg2,  __private int timesteps,
                        __global volatile float  *restrict g_Q_ping,
                        __global volatile float  *restrict g_Q_pong
                        )
{
    // Outer timestep loop
    #pragma loop_coalesce 2
    for (int step = 0; step < timesteps; ++step)
    {
        // 5 RK steps
        for(int intrk = 0; intrk < 5; ++intrk)
        {
            // compute timestep
            int stepCount = step*5 + intrk;

            // switch the buffer: Ping in EVEN step, Pong in ODD
            __global volatile float *restrict g_Q_in = (stepCount%2 == 0)? g_Q_ping:g_Q_pong;
            __global volatile float *restrict g_Q_out = (stepCount%2 == 0)? g_Q_pong:g_Q_ping;

            // Iterate over the  elements
            for (int k = 0; k < K k++)
            {
                float somevalue = g_Q_in[k];
                ...
                somevalue = somevalue + anothervalue;
                ...
                g_Q_out = someothervalue; // Creates Write-ACK LSU
            }
        }
    }
}
```

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`. As the texttg_resQ write for a element is always scheduled after the read for that element in the `RK` kernel as shown in the kernel execution sequence in Figure 5.4, the operations have no dependencies and can be aliased.

5.2.2 Host code updates

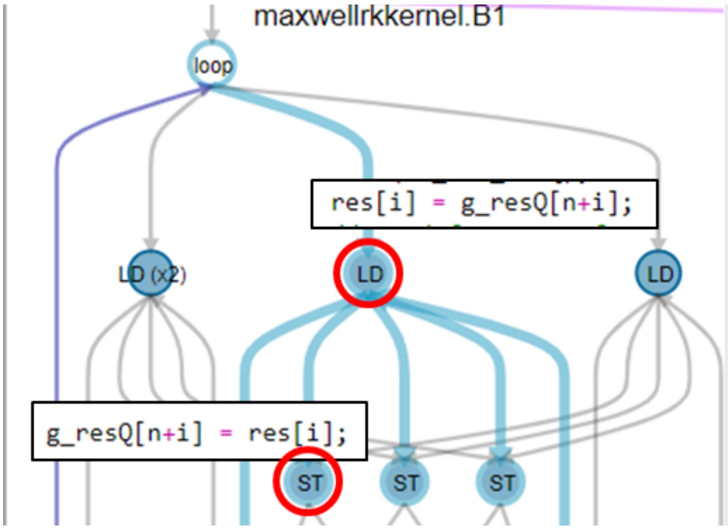
Addition of the FPGA only design required reworking of the host code in order to support all the designs variants with a single host code for performing the evaluation. Apart from the multi design support changes there are also other minor changes done in the code necessary to initialize the memories 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 ad-

Listing 5.8: Aliasing of the `g_Q_xx` buffer to avoid memory dependencies identified by the Intel OpenCL compiler

```
__kernel void kernelName(__private int K,  __private int arg2, __private int timesteps,
                         __global volatile float  *restrict g_Q_ping,
                         __global volatile float  *restrict g_Q_pong,
                         __global volatile float  *restrict g_Q_ping_2,
                         __global volatile float  *restrict g_Q_pong_2
                         )
{
.....
    // switch the  buffer: Ping in EVEN step, Pong in ODD
    __global volatile float *restrict g_Q_in = (stepCount%2 == 0)? g_Q_ping:g_Q_pong;
    __global volatile float *restrict g_Q_out = (stepCount%2 == 0)? g_Q_pong_2:g_Q_ping_2;
.....
}
```



```
res[i] = g_resQ[n+i];
g_resQ[n+i] = res[i];
```

Figure 5.4: `RK` kernel execution sequence showing read of `g_resQ` followed by write

dition of FPGA only design, three variants of the kernels are available viz. `MIDG2 MPI FPGA` (SingleFPGA), FPGA-to-FPGA communication using IO channels with host synchronization (MultiWithHost) and FPGA only design (MultiFPGAOnly). Initially different versions of the same configuration code were used by including or excluding design specific changes using preprocessor directives or by selecting different files for compilation. This 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 5.5. 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.

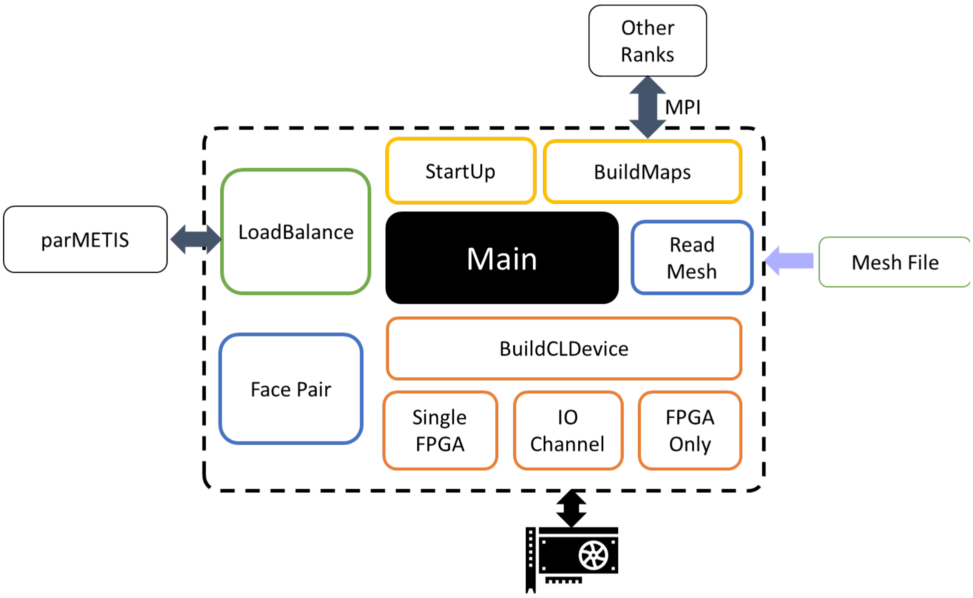


Figure 5.5: Module structure of the the host application after restructuring showing three separate sub-modules to handle OpenCL device configuration

To achieve this restructuring additional structures were included which allows to group the variables into functionality based groups. A important structure to hold 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.

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 BittWare 520N

The BittWare 520N boards using Intel Stratix 10 FPGA provides 4 external memory channels to access the global memory. Each of the 4 external memory channels 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

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

```
// Structure to hold the interface functions
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;

// Initialization example in the sub-module
static tclIntfHandlers intfHandlers =
{
    initKernelArgs,
    FreeKernelMem,
    createBuffers,
    writeToBuffers,
    runKernel,
    getKernelData
};

// Access function to get the pointer to the interface
// handlers structure variable
tclIntfHandlers* XXXX_getDeviceIntfHandlers(void)
{
    return &intfHandlers;
}
```

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.

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

The MPI implementation was designed and optimized for the BittWare 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 BittWare 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 use channel 3 for the g_resQ buffer instead of channel 2. As channel 3 is highly utilized by other memory buffers as well and channel 2 is will only be occupied in the even or odd iterations due to buffer switching, this change reduces the stalls for the g_resQ buffer read and write. The final channel assignment used for base MIDG2 MPI FPGA design, the MIDG2 FPGA IO channels and FPGA only design is shown in table 5.1 and the definitions of the symbols used for size computation as below.

Table 5.1: Updated memory channel assignment for the buffers in MIDG2 MPI FPGA, the MIDG2 FPGA IO channels and FPGA only design

Buffer	Size	MPI FPGA	FPGA ONLY/IO CHANNEL
Q1 Ping	K* BSIZE * pNfields	1	1
Q1 Pong		1	1
Q2 Ping		2	2
Q2 Pong		2	4
surrhsQ		3	-
volrhsQ		4	-
ResQ		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

```
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 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 byte 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 structure but as explained in section 4.1, the writes to g_partQ buffer were identified as dependent leading to 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

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

```
mesh->parNtotalout = 0;
int entries;
for (p2 = 0; p2 < nprocs; ++p2)
{
    entries = skP[p2] * p_Nfields;
    if (fFcdesign)
    {
        float value;
        if (entries%16 != 0)
        {
            value = (float)entries/48;
            entries = ceil(value)*48;
        }
        if (p2 != procid && entries > 0)
            start[p2] = mesh->parNtotalout;
    }
    mesh->parNtotalout += entries;
}
```

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 is also required as the the reads from the buffer are at 24 byte aligned as explained in above. To achieve this, modifications were done in the `BuildMaps3d` function which collects the shared elements information and computes the indexes of the elements in the `g_Q` buffer. The individual partitions are aligned to 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.

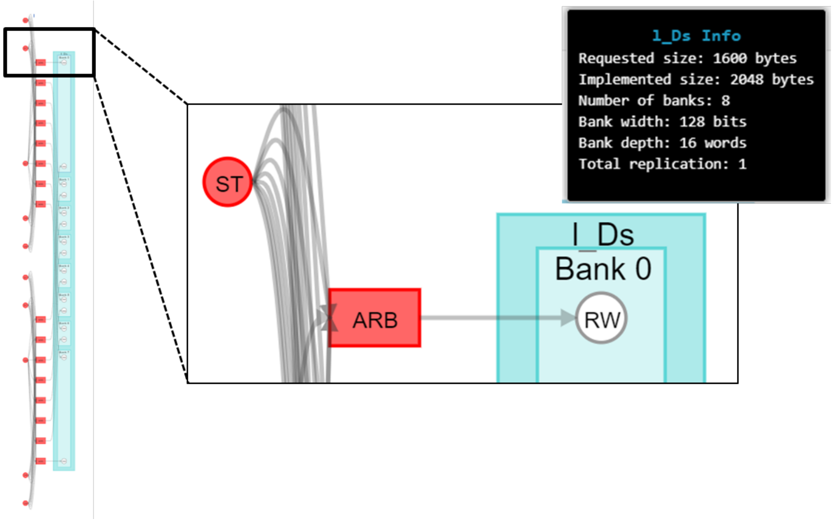
Rearrangement of elements in `g_Q`

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 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 2.6.

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 BittWare 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 5.6. This resulted in a decrease of performance due to stalled local memory reads which are expensive and causes pipeline stalls.

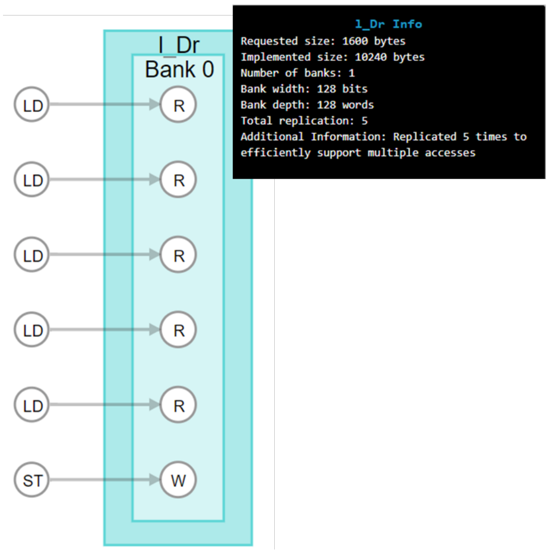
The kernels were analysed by looking at the synthesized reports and it was noticed that the 18.1.1 version of the Intel OpenCL compiler, automatically unrolled the loops which were



```
l_Ds Info
Requested size: 1600 bytes
Implemented size: 2048 bytes
Number of banks: 8
Bank width: 128 bits
Bank depth: 16 words
Total replication: 1
```

Figure 5.6: Local memory structure with arbitration units (ARB)

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 5.7



```
l_Dr Info
Requested size: 1600 bytes
Implemented size: 10240 bytes
Number of banks: 1
Bank width: 128 bits
Bank depth: 128 words
Total replication: 5
Additional Information: Replicated 5 times to
efficiently support multiple accesses
```

Figure 5.7: Local memory structure after addition with pragma unroll 1

Chapter 6

Evaluation

This chapter presents the details of the evaluation structure and the results captured. The first section focuses on the comparative evaluation of the MIDG2 implementations developed in the thesis and the reference base `MIDG2 MPI FPGA` implementation to understand the benefits of the optimization due to introduction of IO channels. In the second section, we look at the analysis results which focuses on understanding the sequence of operations to identify the bottlenecks and future improvements of the design.

6.1 MIDG2 Execution time evaluation

As explained in the previous chapters, the IO channels are integrated into the distributed `MIDG2 MPI FPGA` implementation and additional optimization are performed to remove the host interface and reduce the synchronization overheads in the design. To evaluate the benefits of the changes, the designs are evaluated in this section in terms of the speedup of the processing phase of the MIDG2 application. The execution time of the processing phase contains the computation of the electric and magnetic fields using the DG method and communication for exchanging the shared elements data for the distributed system. First, details of the evaluation setup is given followed by the evaluation of execution time and speedup analysis. The second part presents benefits of the IO channels by evaluating the designs in terms of bandwidth by artificially increasing the data transfer sizes. In the last, global memory bandwidth changes in the kernels captured using the profiling interface of the Intel OpenCL FPGA SDK would be presented.

6.1.1 Evaluation Setup for MIDG2

The experiments like with topologies are conducted on the same nodes in the FPGA partition of the Noctua cluster. The kernels for the experiments are synthesized using Intel FPGA SDK for OpenCL v18.0.1 Pro (build 261) and v18.1.1 Pro (build 263).

The execution time evaluation of the MIDG2 application is done by using 8 different mesh sizes which starts from 1 thousand elements till 200 thousand elements. This provides the necessary problem size variation for the evaluation and comparing the speedup capabilities over the different element sizes. The execution time evaluation is done for polynomial order of `p = 3` which is the default order of the MIDG2 application. Other orders were not evaluated as additional efforts were required to configure the unrolling factors in the kernels which was not feasible in this thesis. The table 6.1 lists the values of the constants dependant on the polynomial order which affect the unrolling factor in the kernels, memory sizes of the buffers within the kernels and host as well as the arithmetic intensity of the kernels.

This thesis utilizes the same unrolling factors proposed in [17] for the single FPGA implementation but the arithmetic intensity of the compute kernels are improved as the memory write operations to store the right hand side field values **rhsE** and **rhsH** in both the kernels are removed. The **rhsE** and **rhsH** values are instead communicated via Intel OpenCL channels as explained in the section 4.1. The detailed arithmetic intensity computation for the `VOLUME` kernel in `MIDG2 FPGA` implementation is summarized in [**TABLE I**, 17]. The removal of **rhsE** and **rhsH** removes the $N\dot{6}$ float elements written into the memory reducing the total floats to 129. As the number for floating point operations remains the same the arithmetic intensity of the `VOLUME` kernel is doubled to 31 operations/byte for `p=3`. The `SURFACE` kernel also has improvements in the arithmetic intensity though it is not as significant as `VOLUME` kernel since **rhsE** and **rhsH** contributed to only 13% of the IO operations. The total number of floats read/written now are 760 giving a arithmetic intensity of 3.96 operations/byte. The `RK` kernel has low arithmetic intensity as very few arithmetic operations are performed which is now 0.3 operations/byte compared to 0.25 operations/byte in the `MIDG2 FPGA` implementation.

Table 6.1: Constants value at `p = 3`

Constant	**Computation**	**Value at p = 3**
VOL_UNROLL	p_Np	20
SURF_UNROLL	p_Nfp	10
p_Nfp	((p + 1)*(p+2)/2)	10
p_Np	((p + 1)*(p+2)*(p+3)/6)	20
p_max_NfpNfaces_Np	max(p_Nfp*p_Nfaces,p_Np)	40
BSIZE	(PFAC*((p_Np+PFAC-1)/PFAC))	20

The execution time computation in the MIDG2 application is done using `clock_gettime()` Linux time API. The same time computation is added for all the designs to measure the processing time in each of the designs. The processing time for each execution of the host application includes

- Computation time for computing field values for `K` mesh elements executing for `timestep = 35` and `finalTime = 0.005`. This along with 5 Runge-Kutta stages per timestep gives 135 iterations. The `finalTime` and the `timestep` values are fixed in the application instead of computing using the mesh size to fix the number of computation iterations for each mesh size. This allows to identify the effects of the `HOST` - FPGA communication latency.
- Communication time for transferring `parNtotal * 4` bytes per iteration using either IO channels or MPI+PCIe.

The host application restructuring made the benchmarking simpler as it is simpler to separate common functionalities such as execution time computation in the top level files. The execution time is computed using `clock_gettime()` to capture the start time and end time of the kernels and communication. The measurement is done in the top level `BuildCLDevice` which simplifies the computation for all the designs.

The restructuring also enables to write shell scripts which are used to invoke the application with different parameters to test different designs and variants with the same binary. For the evaluation the host application is modified to include repeated execution of the kernels for the same mesh for n number of times. This allows to repeat the tests with same mesh for multiple times and use the arithmetic mean of the execution times of all the runs for the comparison purpose. All the tests performed repeat the tests for 30 times for each mesh size.

Every run, the application first configures the FPGA device using `BuildKernels`. The configuration involves identifying the platform and the device to be used, creating context for the device, programming the FPGA device with the provided kernels binary, creating OpenCL buffers in device memory and then writing the buffers with initial data. This is followed by setting the parameters in the kernels. Once the configuration is successful, the application executes the kernels by calling `run_kernels` routine which invokes the `RunKernel_xxx` routine that enqueue the kernels and perform the computation. Depending on the implementation, the timestep iterations are controlled in the `host application` or the FPGA. The execution time for each run is computed separately and accumulated after each run. After completing all the runs, the computation is completed and the accumulated execution time is written into a output CSV file. The output CSV contains mesh size, order of the execution, min, max and avg execution time for the processing, size of data communicated by the node, the nodal and the L2 Norm errors computed from the final field values.

The evaluation for execution time is done for 5 different implementations. Table 6.2 lists all the evaluated designs and the variations. along with the names used in the following text, tables and plots for the variants. The first design is the base `MIDG2 MPI FPGA` (section 2.5) design running with 2 distributed single FPGAs (`MPI_N2`) or 4 single FPGAs (`MPI_N4`). The next design evaluated is `MIDG2 FPGA IO Channel` (section 4.1) design exchanging data using IO channels using within node topology using 1 channel (`WNIO`) or 4 channels (`WNIO4CH`) and `fully connected` topology (`FCIO`). In `XXIO` designs `host application` performs the synchronization and time-stepping. The next design evaluated is the `FPGA Only` (section 5.2) design in within node (`WN`) and `fully connected` topologies (`FC`). These two variants do not use `host application` for synchronization. Each of the design was synthesized with both the toolchains

Table 6.2: MIDG2 designs with the variations used to perform the run time evaluation

Design	FPGAs used	Mem flags	Added latency	Synthetic Test	Label
MIDG2 MPI FPGA	2	Yes	NA	No	*MPI_N2*
	4	Yes	NA	No	*MPI_N4*
	2	No	NA	No	*MPI_N2_nf*
	4	No	NA	No	*MPI_N4_nf*
Within Node with 1 IO Channel	2	Yes	NA	No	*WNIO*
	2	No	NA	No	*WNIO_nf*
Within Node with 4 IO Channels	2	Yes	NA	No	*WNIO4CH*
	2	No	NA	No	*WNIO_4CH_nf*
Fully connected with IO Channel	4	Yes	NA	No	*FCIO*
	4	No	NA	No	*FCIO_nf*
Within Node FPGA Only 1 Channel	2	Yes	Yes	No	*WN*
	2	Yes	No	Yes	*WN_nolat*
	2	No	Yes	No	*WN_nf*
	2	No	No	Yes	*WN_nolat_nf*
Within Node FPGA Only 4 Channels	2	Yes	Yes	No	*WN4CH*
	2	Yes	No	Yes	*WN_4CH_nolat*
	2	No	Yes	No	*WN_4CH_nf*
	2	No	No	Yes	*WN_4CH_nolat_nf*
Fully Conneted FPGA Only	4	Yes	Yes	No	*FC*
	4	Yes	No	Yes	*FC_nolat*
	4	No	Yes	No	*FC_nf*
	4	No	No	Yes	*FC_nolat_nf*

to evaluate the difference in the performance achieved with the newer toolchains. Additional variation for the global memory interconnections available in the Intel FPGA SDK OpenCL v18.1.1 as explained below was also included (`XXX_XX_nf` suffixed variants in the Table 6.2). As the latencies included in the FPGA only designs contribute a large time to the execution of the meshes with lower sizes, another set of tests were performed for some of the designs by setting the latency to only 1 cycle (`XXX_XX_nolat` suffixed variants in the Table 6.2). These variants do not produce correct results and are used only to understand the impact of the latency to the `FPGA Only` design.

To perform the test with all the designs listed in the Table 6.2, the kernels for each design were synthesized with both versions of the tool. To disable the ring interconnect for global memory and duplication of the store ring which are explicitly forced by using the flags `-global-ring -duplicate-ring` were removed by removing the flags in the second synthesis. Removing the flags lets the Intel FPGA SDK for OpenCL offline compiler to choose the optimal global interconnect topology as explained in [**section 7.16 and 7.17**, 14]. The kernel binaries for each of the synthesis are placed in separate directories to be available for use at the same time. The table 6.3 lists the frequencies of the synthesized designs and table 6.4 lists the resource utilization summary of the designs with both toolchains.

Table 6.3: Frequencies for the synthesized design

Design	**Fmax(MHz)**		
	18.0.1	**18.1.1**	**18.1.1_nf**
MPI	312.3	283	288.68
WNIO	281	287.35	278.86
FCIO	262.5	223.36	244.91
WNIO4CH	223.01	257.2	257.26
WN4CH	254.16	248.75	231.42
WN	265.32	276.93	254.84
FC	236.4	239.46	237.41

Table 6.4: Resource utilization in (%) of the designs in each toolchain version

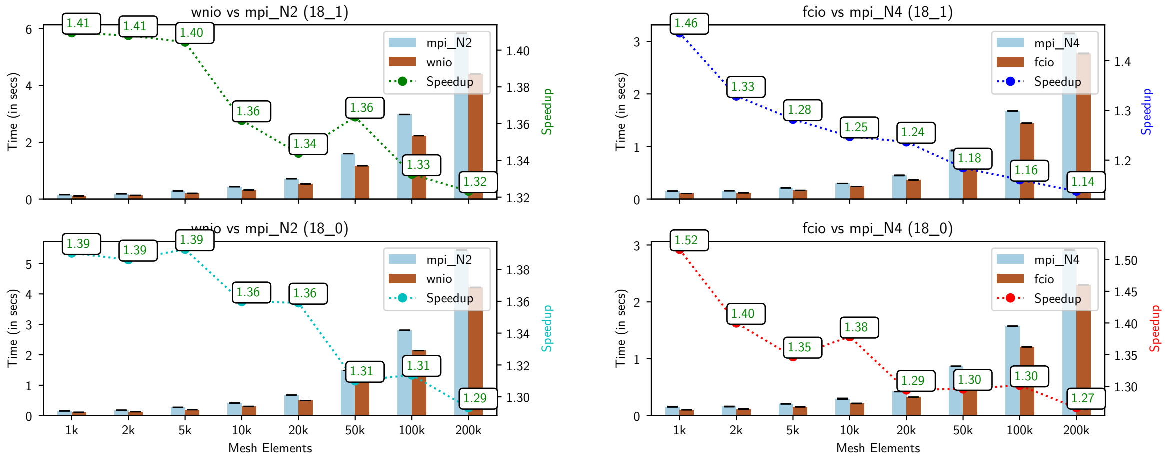
Resource	**Toolchain**	**Design**						
		MPI	WNIO	FCIO	WNIO4CH	WN4CH	WN	FC
Logic (in %)	18.1.1	26	25	28	27	34	33	36
	18.0.1	34	35	43	41	41	36	43
DSP (in %)	18.1.1	13	12	12	12	12	12	12
	18.0.1	13	12	12	12	12	12	12
RAM (in %)	18.1.1	12	12	14	14	12	11	13
	18.0.1	11	11	13	13	11	10	12

6.1.2 Speedup Evaluation

This section presents evaluation results for the execution time evaluation in terms of speedup of the execution time for different mesh sizes.

MIDG2 MPI FPGA vs IO Channels

The first modification done as part of the thesis is the addition of IO channels. The aim with addition of IO channels is to reduce the communication overhead due to PCIe transfers involved in exchanging the non-shared elements between the nodes. As the communication is overlapped with the execution as shown in the sequence graph in the Figure 4.3, the expected improvements are smaller than the improvements of raw memory bandwidths from the synthetic tests with the topology prototypes. Figure 6.1 shows the plot with speedup values achieved for the MIDG2 FPGA IO channels design over the MIDG2 MPI FPGA designs. The within node topology has higher speedup values then the fully connected designs for bigger mesh sizes. The speedup for the smaller mesh sizes (<10k) are higher in all the variants. As the ratio between contribution of computation time and communication time to the overall execution time for the smaller meshes is small, higher speedup is expected. The IO channels having higher bandwidth and lower latency reduces the communication time reducing the effects of higher delays in communication which is present in the MPI based communication. On the other hand, the large computation time for large meshes compensates for delays in communication. The speedup values for the within node topology vary between 1.30 to 1.40. This could be explained as the data sizes in a two node design would be larger and have more effect on the overall execution time then in the 4 node designs. Also, as it was noticed in the topology evaluation, the 1 channel topology is able to achieve higher bandwidth for smaller data sizes which could be another factor. Comparing the tool chain variation, it can be noticed that the within node design perform similar in both the versions whereas for fully connected designs the 18.1.1 toolchain has lower speedup values which goes to a minimum of 1.14 for the 200k compared to 1.27 which is around 10% faster. The FCIO variant with 18.1.1 toolchain has a clock frequency of 223.36 MHz compared to 262.5 MHz in 18.0.1 toolchain which could be one of the reasons for lower speedup values for higher mesh sizes.

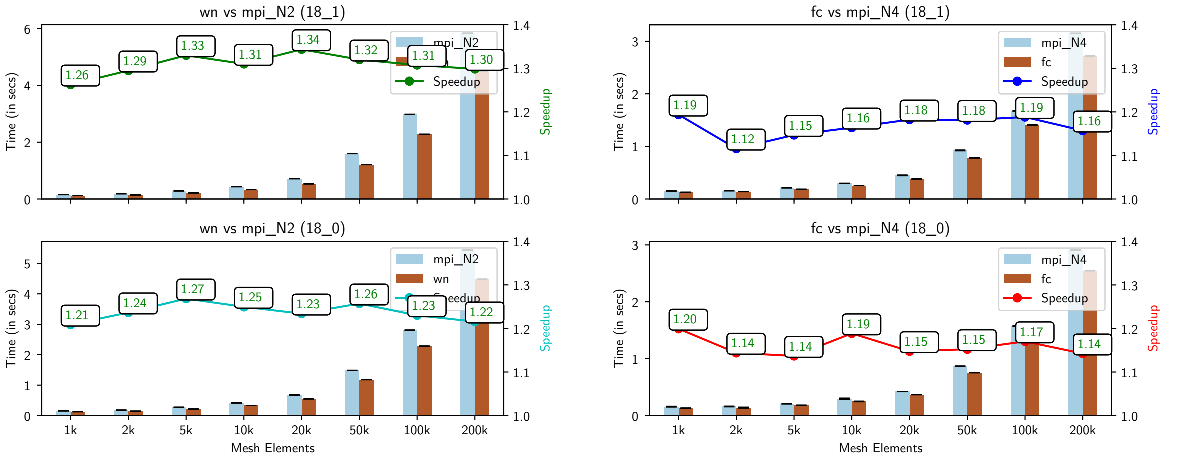


(a) (b)

Figure 6.1: MIDG2 MPI FPGA vs MIDG2 FPGA IO Channels designs

MIDG2 MPI FPGA vs FPGA ONLY

As explained in chapter 5, to avoid the host synchronization which added some amount of overhead to the processing, the second modification removed the host synchronization completely. These designs were implemented to utilize the full potential of the FPGA without the host interference. Figure 6.2 shows plot of speedup of `FPGA only` over `MIDG2 MPI FPGA` designs varying over different mesh sizes. Overall the speedup values for **`fully connected (FC)`** topology is lower than the **`within node (WN)`** topology in both the toolchains similar to the **`IO channels`** design. Comparing the effects of toolchain variation, for both the designs, the 18.1.1 version performs slightly better overall and especially for higher data sizes unlike the IO channels where designs with 18.0.1 toolchain perform better.



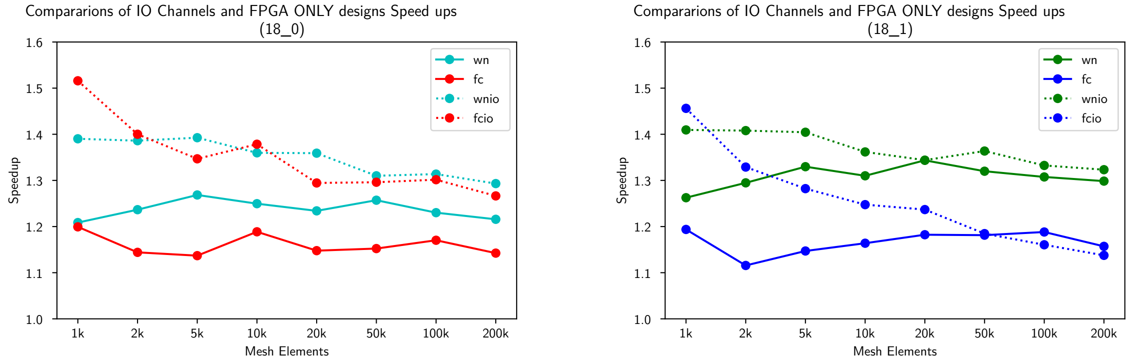
(a) (b)

Figure 6.2: MIDG2 MPI FPGA vs MIDG2 FPGA only design

Comparing the performance of the `FPGA Only` design with `FPGA IO channels` design it can be seen that the further optimizations done in `FPGA Only` have no benefits and produces smaller speedups than the `FPGA IO channels` designs over the base `MIDG2 MPI FPGA` design. Figure 6.3 shows the plot comparing the speed values achieved for the IO channels designs and the `FPGA Only` designs with respect to the `MIDG2 MPI FPGA`. Both `WN` and `FC` variants of `FPGA Only` have significant decrease in the speedup values. The impact of the changes is much higher in the case of the 18.0.1 toolchain where for `fully connected` topology, there is a decrease in speedup of 21% with 1k mesh size and 10% for 200k mesh. With 18.1.1 version the deviations are only higher for smaller mesh sizes and converge towards the higher mesh sizes which could be because of the use of fixed latency time within the kernels which is around 150000 cycles (approx. 75ms at 270MHz) which could affect the execution time of smaller mesh sizes.

Effects of global memory interconnect flags

The 18.1.1 toolchain version provides additional memory optimization flags which control the global memory interconnections to the FPGA fabric. As explained previously, the designs were also synthesized with the flags disabled to see the effects on the performance. The plots in Figure 6.4 show the comparison of the speedup of the IO channel and FPGA only designs with MPI MIDG2 with and without memory flags. For FPGA only design the Fully connected topology have similar speedup and show no effects of the flag but in case of the within node topology, disabling the flag reduces the speedup marginally for all the mesh sizes but difference is very small and could be attributed to variations. For the IO channels the, the within node design has no effect whereas the fully connected design has benefits for all mesh sizes for about 10% increase in speedup.

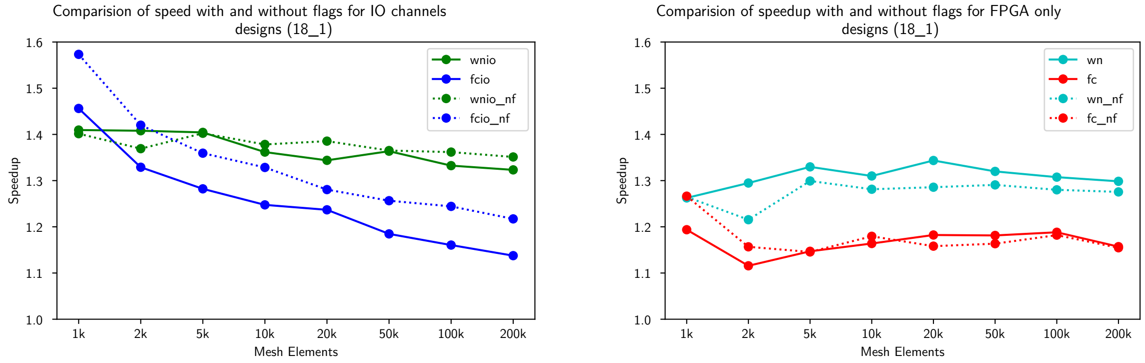

Compararions of IO Channels and FPGA ONLY designs Speed ups (18_0)

Compararions of IO Channels and FPGA ONLY designs Speed ups (18_1)

(a)

(b)

Figure 6.3: MIDG2 FPGA IO channels vs MIDG2 FPGA only design


Comparision of speed with and without flags for IO channels designs (18_1)

Comparision of speedup with and without flags for FPGA only designs (18_1)

(a) MPI vs IO Channel

(b) MPI vs FPGA Only

Figure 6.4: Effects of memory optimization flags in 18.1.1 tool chain

6.1.3 Scaling performance over multiple FPGAs

The scaling performance for a system gives the understanding about the efficiency of the design in terms of resource utilization. While scaling an implementation over multiple resources, often the bottleneck is the communication overhead which increases with more no of resources. As the communication overhead is reduced with the IO channels by allowing FPGAs to communicate directly, strong scaling of the system is possible. The figure 6.5 shows the plot with execution time speedup scaled over multiple FPGAs for a 100k mesh to compare the scaling capability of the different FPGA designs. The `MIDG2 MPI FPGA` design which can be scaled easily over multiple FPGAs is executed with 1 to 10 FPGAs in increments of 2. The `IO channels` design only support 2 and 4 FPGAs in the two topologies and the best designs identified in the previous sections for each of the variants are used for the comparison.

We can see that speedup values of both the `IO channels` designs are close or above the linear speed line in the plot whereas the `MIDG2 MPI FPGA` design diverges after 2 FPGAs with decrease in efficiency as more FPGAs are used. The high efficiency of the `IO channels` shows the clear advantage of using the `IO channels` over the MPI+PCIe communication to achieve strong scaling for large mesh sizes with linear speedup.

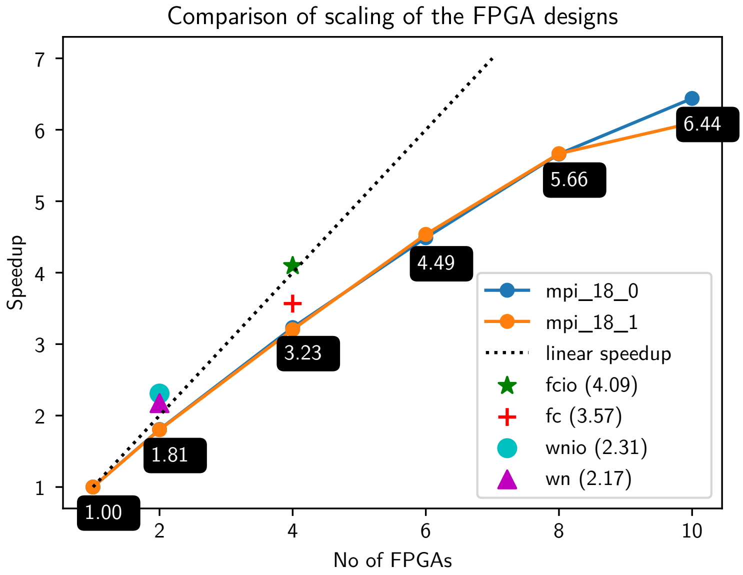


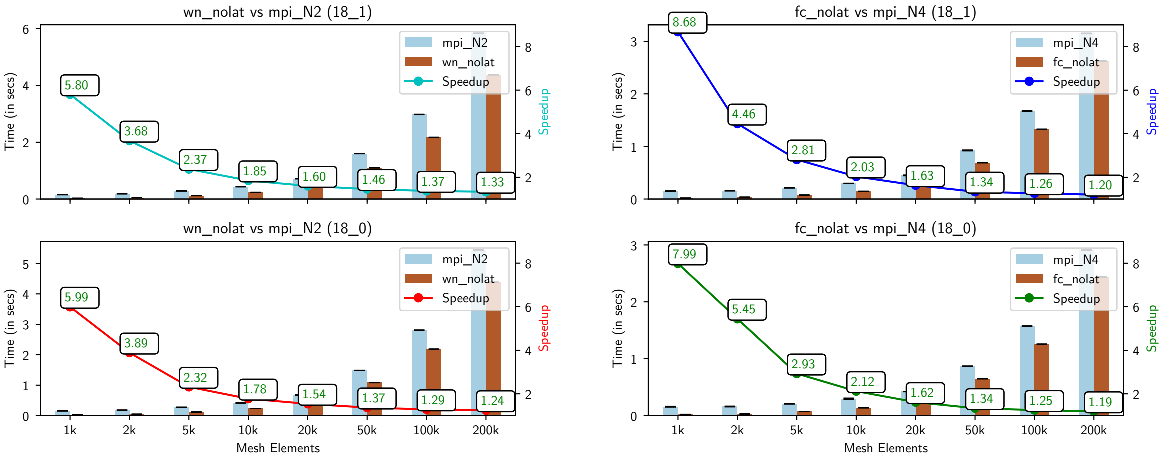
Figure 6.5: Comparison of scaling the FPGA designs. `MIDG2 MPI FPGA` design is scaled over 2,4,6,8 and 10 FPGAs. The `IO channels` and `FPGA Only` design uses 2 (`WN` and `WNIO`) and 4 FPGAs (`FC` and `FCIO`)

6.2 Detailed Analysis of the results

Previous sections present a analysis of the captured evaluation data to compare the performance of the design implemented in this thesis. It was expected to have only smaller improvements with the optimization done in the final `FPGA only` as the optimization only target to eliminate the unnecessary overheads in the design. The analysis of the data show that the optimization didn't improve the performance and have instead decreased the performance for few variants. This section presents a detailed analysis of the results with additional experiments done to identify the bottlenecks and highlight them for future improvements.

6.2.1 Removing the latency

As explained in the section 5.1.1, a latency was required in the kernels to avoid the overlapped write and read of the shared global memory. The required latency was considered to be small within 100 to 1000 cycles but during testing a value of 150000 cycles was found to be the suitable one. As this amounts to a huge value of approx 75ms at 270MHz, further analysis was done. To understand the impact of the latencies in the `FPGA only` design, the `host application` was configured to allow setting up the latency count at the execution time. This is used to set the latency to 1 and run the tests again to see the impact of the latency. The field values produced by this change are not accurate, as the memory operations are not overlapped and corrupts the memory. The tests were also done to understand the potential of the FPGA design in absence of the additional delay. The plot 6.6 shows the speedup values for each mesh size for the no latency run. The plot shows that for smaller mesh sizes very high speedup values of 8.68 for the `fully connected` topologies is possible. The speedup values for the `fully connected` topologies is higher than within node design which could be explained due to smaller communication times for the parallel data transfers. This results in lesser synchronization time delays between the kernels which still use channels to synchronize but with no latency. For higher data sizes the



(a) (b)

Figure 6.6: Synthetic tests of `MIDG2 FPGA only` design with latency set to one cycle count. The MIDG2 application produces wrong results in this tests as memory operations are overlapped

benefits diminish completely giving speedup values similar to runs with latency. This is due to the higher computation times for bigger meshes. As the mesh size increases, the kernel spends more time for computation. The main benefit of the FPGA only design comes from removal of the overhead due to host interactions per iterations. With higher mesh sizes the overhead only contributes a very small fraction of the execution time per iteration and is not significant for the overall execution making the benefits of removal of synchronization minimal.

These synthetic tests show that there is a possibility to further improve the performance of the `FPGA Only` designs with the smaller mesh sizes. It also shows that an unusually large delay is required by the memory controller to complete the memory operations. As the Intel FPGA SDK for OpenCL currently does not provide any global memory synchronization method for the kernels it is not possible to achieve higher performance with the implemented kernel structures.

6.2.2 Bandwidth evaluation with synthetic data sizes

The execution time evaluation shows that there are possible benefits of using the IO channels in terms of improved communication for atleast smaller mesh sizes. With bigger mesh sizes the benefits of the improvements are less prominent as the longer computation part runs simultaneously. To better understand the benefits of the IO channels, an additional set of tests were designed. The aim of these tests is to simulate higher sizes of data transfer in application and compare the bandwidth performance and the execution time in this case. These tests would help to estimate the benefits which could be gained in an application which require large data transfers between multiple nodes and suffer from large communication overhead of the standard HPC networks.

To achieve higher data size data transfers, the configuration for partitioning the mesh was updated to partition meshes at tetrahedral boundaries instead of triangular surface boundaries. This is done by updating the `ncommonnodes` value to 4 from 3, which is the parameter to control the number of vertices of the element to be used for partitioning. The partitioning with tetrahedral is inefficient as the faces of the tetrahedron are the shared entity between the elements. This partitioning configuration produces large partition with upto 80 to 90 percent of elements shared between the partitions. The size of the data shared in both the designs for a 2 node and 4 node system is capture in 6.5

Table 6.5: Data sizes of the data transfers between then nodes for the two different values of ncommonnodes and number of nodes used

Elements	data size 2 node(in KB)		data size 4 Node(in KB)	
	ncom = 3	ncom = 4	ncom = 3	ncom = 4
1052	16.18	219.14	14.44	160.55
1978	23.68	405.71	22.41	308.91
5235	43.6	1131.1	43.04	829.46
10420	83.44	2257.97	76.88	1702.04
19982	137.82	4374.85	118.79	3286.06
51170	247.5	11234.3	231.66	8520
102045	388.83	22602.43	370.6	17072.23
203163	608.21	45163.6	595.13	34011.8

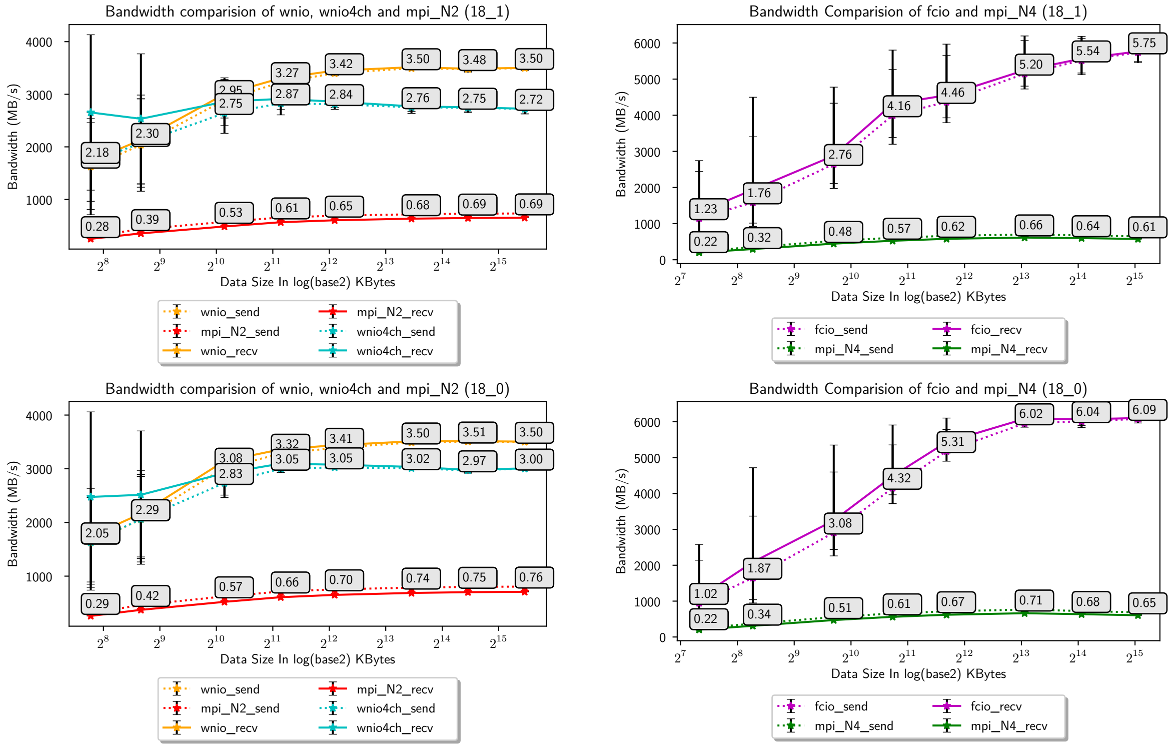
Bandwidth Measurement

The change in the `ncommonnodes` produce correct error values for the `MIDG2 MPI FPGA` design but for the IO channels design, wrong error values are produced. This could be due some alignment issues, but the issue was not analysed in detail due to time limitation and as it just to simulate higher data transfers and would not be used in a real system. The measurements for the bandwidth were taken with `MIDG2 MPI FPGA` and the IO channels designs only. As both the designs have the similar execution and synchronization sequence controlled by `host application`, comparison of these designs is straight forward. Also, the bandwidth computation in FPGA only design is quite complex as there are no host interactions per iterations and kernel waiting time and active time is hard to predict from the profile information.

The bandwidth measurement for the IO channels design is done separately for the send and receive. As the execution time of the `partial_send` and `partial_recv` depends on the global memory performance and the availability of communication partner on the channel, the measurements are done separately. Each of the kernel have different global memory access pattern as well as the stalls on the channels can vary depending the kernel on the other FPGA, the actual bandwidth throughput of the channels in send and receive direction would differ and measuring them separately can be used to understand it better. The measurement of the bandwidth with `IO channel` designs are simple. The kernel execution time for `partial_send` and `partial_recv` kernels is computed by requesting the `CL_PROFILING_COMMAND_<START/END>` profile counters as done in the prototypes. The communication times are then computed and stored in the `profileinfo` buffers. The bandwidth measurements for the `MIDG2 MPI FPGA` implementations is done using the `clock_gettime()` by measuring the total communication time for send and receive and computing the bandwidth similar to bandwidth measurements for the topology evaluation. The computed time is store in the profile information structure which is processed at the end of every run.

Analysis of the results

The plots in the Figure 6.7 show the comparison of the average bandwidth values recorded for the MPI and the IO designs. Figure (a) shows the bandwidth for 2 nodes whereas (b) shows for the 4 node designs. The first thing which is clearly visible is the huge difference in the bandwidth values for the MPI and IO channels design which proof the higher efficiency of the IO channels in real application scenarios as well. The MPI reports a maximum bandwidth of 0.76 GB/s for the 200k Mesh in 2 node design with 18.0.1 toolchain. which is only 25% of the available bandwidth, whereas the within node design is able to utilize the 70% of the available bandwidth reporting a maximum bandwidth of 3.50 GB/s. The Fully connected design as expectation have higher bandwidth values due to utilization of 3 channels though the peak bandwidth of 6.09 for 200k Mesh is only 40% which could be due to the global memory access limitations. As all the kernels compute parallelly now, the load on the global memory is higher for handling the requests which would lead to stalls and explain the bahaviour. The 4 channel design performs



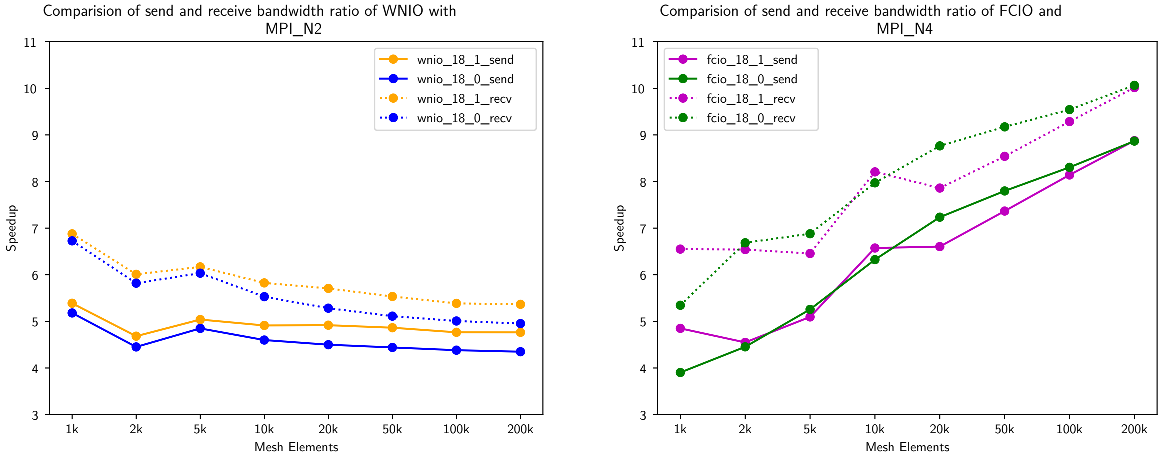
(a) 2 node designs

(b) 4 Node Designs

Figure 6.7: Average bandwidth values for two node topologies `wnio`, `wnio4ch` and 4 node topology `fcio` and designs

slower than the single channel design. As identified in the topology testing the, performance of 4 channel design is affected due to the wider global memory access which cause higher stalls. In the actual application, with other kernels working simulataneously, the effects of the global memory has increased leading to bandwidths slower than the single channel implementation.

The plots in Figure 6.8 compare the bandwidth speedup of send and receive seprately. The receives is both IO channels design have significantly higher speedup values then the sends. The difference can be accounted to the sharing of global memory buffer in the send and hence the memory band with other kernels. The `partial_send` kernels reads the `g_Q` memory buffer to get the shared data values. With the addition of channels between `RK` and `VOLUME/SURFACE` kernel, the concurrent access of the `g_Q` increases which would cause stalls and delays in the send kernel leading to slower sends. Looking at the plots it can also be clearly seen that performance of design synthesized with 18.0.1 toolchains is lesser as the MPI design performs better with the 18.0.1 toolchain reporting higher bandwidth values. Although this behavior can only be seen in the within the node topology. The `fully connected` topology has similar performance in both toolchains.



(a) 2 node designs

(b) 4 Node Designs

Figure 6.8: Send and receive bandwidth comparison for `within node` and `fully connected` topologies with the simulated high data transfers between the FPGAs using inefficient partitioning of the mesh

Execution time comparision

The execution time in the synthetic tests contains a larger fraction of communication time as ratio of the shared and non-shared elements is 4:1. The plot in Figure 6.9 show the comparison of the execution time speedup for the `IO channel` designs over the `MIDG2 MPI FPGA` design used for

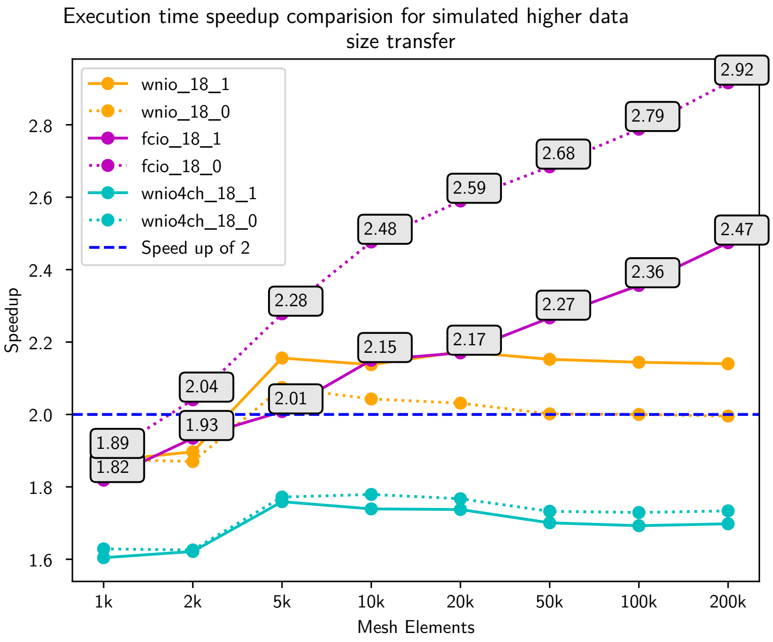


Figure 6.9: Execution time Speedup comparison for simulated higher data size transfers

bandwidth evaluation. A maximum speedup value of 2.92 is reported for the `fully connected` designs with both within node and `fully connected` topology having speed of 2 and above for higher mesh sizes. This shows the potential of using the IO channels with higher data transfer sizes which was not clear with the execution time evaluation. The within node 4 channel design has speedups below 2, mostly due to the global memory congestion in the send part as noticed with the earlier analysis. The `fully connected` design synthesized with the 18.0.1 version is the only 18.0.1 design which performs better. After further analysis it was identified that the difference could be due to addition of write-ack LSU for the `g_resQ` buffer in the the `RK` kernel which is solved in the FPGA only designs by aliasing the memory. The execution time speedup comparison with the large data sizes shows the benefits of using IO channels in applications with very high communication demand compared to the computation. This understanding can be used with MIDG2 application by identifying other partitioning schemes which generates smaller partitions but a higher percentage of shared elements. As the current shared percentage for large mesh sizes is too small (2% to 6%), having a alternative schemes or execution structure which is more communication intensive can benefit a lot from the IO channels of the FPGAs.

6.2.3 Bottleneck analysis in FPGA Only Design

The FPGA only design for both the topologies achive very small speedups for the large mesh sizes even after setting the explicit latency value to 1. The bahaviour of the kernels with larger mesh sizes was analysed further by profiling the kernels using the Intel FPGA Dynamic Profiler for OpenCL. The kernels can be instrumented with performance counters by specifying the `-profile` flag during the synthesis of the kernels. The kernel synthesized with the profile flags captures the performance counters and saves in a file called `profile.mon` in the execution directory. Intel FPGA Dynamic Profiler for OpenCL can be then invoked with the data collected from the kernels to understand the performance bottlenecks. The profiler parses the collected data and displays statistical information collected from local memory, global memory and channel access in the kernels. The main attributes presented in the profile information are listed with thier description in the Table 6.6:

Table 6.6: Description of the profile properties taken from [13]

Property	**Description**
Stall%	Percentage of time the memory or channel access is causing pipeline stalls. It is a measure of the ability of the memory or channel access to fulfill an access request.
Occupancy%	Percentage of the overall profiled time frame when a valid work-item executes the memory or channel instruction.
Bandwidth	Average memory bandwidth that the memory access uses and its overall efficiency. For each global memory access, FPGA resources are assigned to acquire data from the global memory system. However, the amount of data a kernel program uses might be less than the acquired data. The overall efficiency is the percentage of total bytes, acquired from the global memory system, that the kernel program uses.

The initial analysis of profile information suggested that the computation in the `VOLUME` and `SURFACE` kernel were the bottleneck. A high percentage of stalls were reported on the channels feeding data into the `VOLUME` and `SURFACE` kernel as well on the channels which are written from them. To analyse the bottleneck, the structure of the FPGA only kernel was simplified by removing `RK`, `partial_send` and `partial_recv` kernels. The removal of these kernels allows to profile the data flow within the `VOLUME` and `SURFACE` kernels independentaly without any

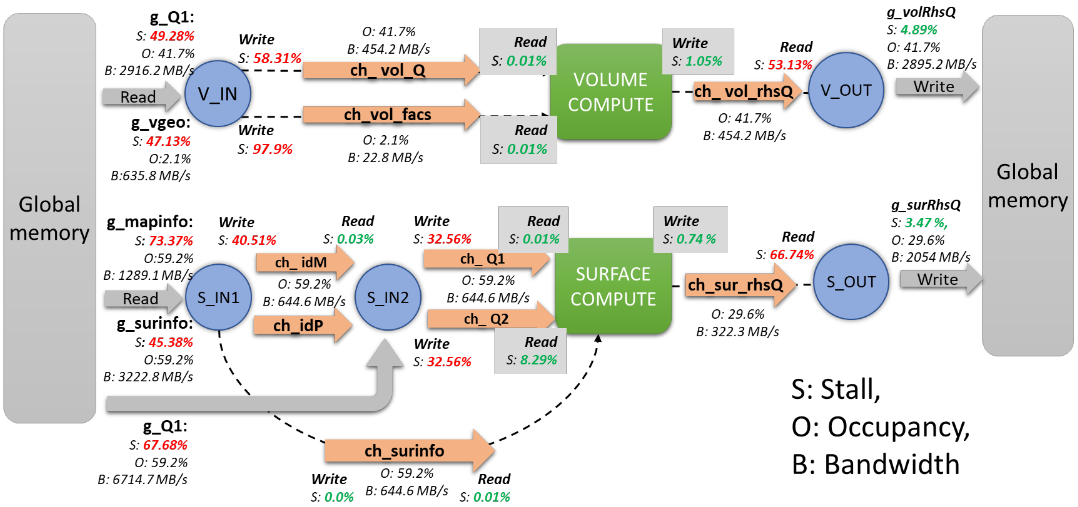


Figure 6.10: Structure of the analysis kernels along with the profile information for the channels and memory access with kernels synthesized with v18.1.1 and ran with 100k mesh for 35 timesteps. The kernel have the timestep loops within the kernels similar to `FPGA Only` design

synchronization overheads. Two additional support kernels were included which read the data written by the `VOLUME` and `SURFACE` kernel and write into the global memory. The timestep logic is kept to simulate similar execution intervals of the kernels. The modified kernels along with the reported stalls, occupancy and bandwidth are shown in the Figure 6.10

The profile information is similar to the FPGA only design confirming that the effect of synchronization structure to the computation is negligible. To analyse the bottleneck, the data movement can be traversed from global memory to global memory via the kernels and channels looking at stalls and occupancy values the each location. Looking at the movement of `g_Q1`, via `VOLUME` kernel, `V_IN` reads the data from the global memory first. As shown, the reads are performed with a stall% of 49.28% at 2916.2 MB/s. This means that the `V_IN` kernel is unable to read data from global memory efficiently and half of the read requests are only finished without delays. The read values are written into the channel `ch_vol_Q` with stall% of 58.31% which is higher than the global memory stall. The data is then read by the `VOLUME` kernel. The low stall% of 0.01 means that the for each read request on the channel, the `VOLUME` immediately gets a data without waiting. At this point, it can be concluded that the data transfer from the global memory to `VOLUME` kernel happens at a rate which is sufficient for the processing capabilities of the `VOLUME` kernel. The high stall% on the write to `ch_vol_Q` from `V_IN` confirm that `VOLUME` is not able to process all the data fed into the channel with the same rate. Though there is a high % of stall on the read from the global memory, the higher stall% for channel would negate the effect of the global memory stall as all the data read from the global can still not be processed by the kernel.

Traversing further, the `VOLUME` kernel writes the processed data into the `ch_vol_rhsQ` with a stall% of 1.05%. The data is read from the channel by the `V_OUT` kernel with a stall of 53.13% and written into the memory at a bandwidth of 2895.2 MB/s with a stall% of 4.89%. Looking at these values, it can be concluded that `VOLUME` is able to write data out without any delays into the channel but `V_OUT` kernel waits for the `VOLUME` kernel half of the time to write something. Also, it can be confirmed that `V_OUT` kernel is able to write all the data read from the channel immediately into the global memory without delays.

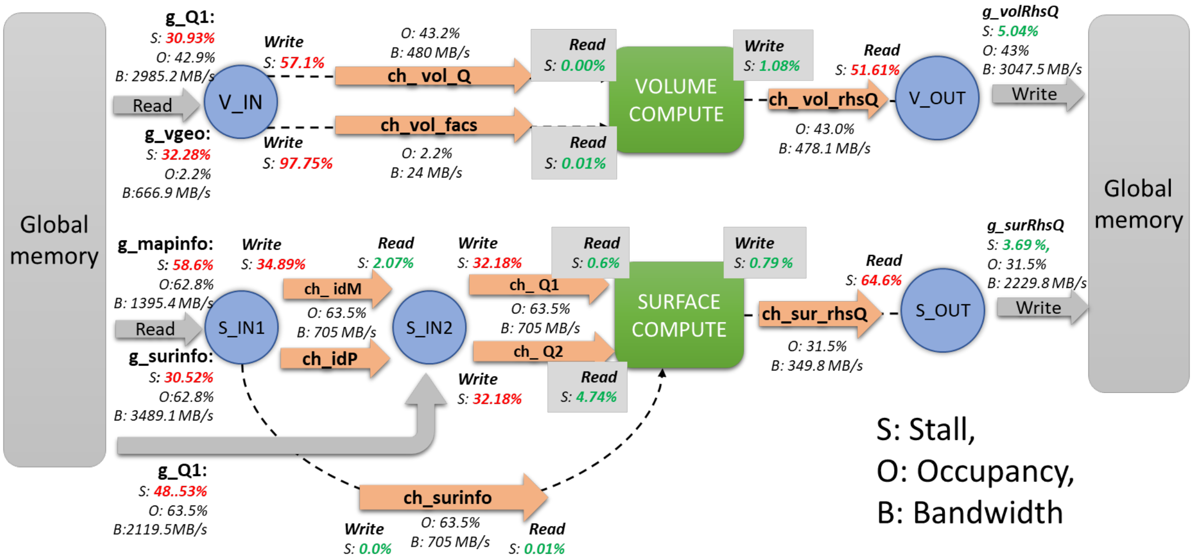


Figure 6.11: Structure of the kernels with the timestep removed along with the profile information for the channels and memory access with kernels synthesized with v18.1.1 and ran with 100k mesh for 35 timesteps

Looking at the occupancy along the path, it can be seen that it is 41.7% at each of the location. From this example traversal, it can be confirmed that bottleneck for the `g_Q` buffer data movement is at `VOLUME` kernel as it is not able to read the data at the rate fed into channel `ch_vol_Q` as well as not able to provide data at the requested rate into the channel `ch_vol_rhsQ`. The data is being processed only 41.7% of the total execution time in the `VOLUME` which can be derived from the occupancy value as that would cause the stall of 58.31% at the `V_IN`.

The values above show that synchronization via the channels does not affect the performance of the kernels and the bottleneck is with the compute kernels. As the timestep loops are included in the kernel, the pipeline structure of the nested loops was not clear from the reports generated by Intel OpenCL compiler. The timestep loops could limit the Intel OpenCL compiler from optimizing the original compute kernel structure due to higher level of nested loops. To confirm this hypothesis another test was performed by removing the timestep loops. Figure 6.11 shows the structure of the kernels which is similar to previous structure as only the loops are removed. We can see that the removal of loops has no effect on the performance of the compute kernels. The kernels still are responsible to stall the pipeline before and after them. We can only see a minor improvement for the global memory access where the stalls values have reduced. The stalls and the occupancy for the channels writing into the compute kernels and reading out of the compute kernels remain similar.

The highlighted values in the Figure 6.10 and 6.11 show the percentage of stall for channels used to read data to be processed into the compute kernels `VOLUME` and `SURFACE` and write data out after processing. The small values of the stalls for both kernels confirm that the computation in the compute kernel is not fast enough to process all the data fed by the pipeline. The `SURFACE` has a larger impact on the pipeline as it is slower out of the two compute kernels. As it can be seen from the occupancy values in the Figure 6.10, the `SURFACE` kernel reduces the occupancy of the pipeline from 60% to 30% as the two sequential inner loops in the kernel are unbalanced. These results show that further improvement of processing in the compute kernel is required to improve performance. This can be achieved by restructuring the kernels to allow more parallel computation to improve the memory and channel occupancy of the complete pipeline.

Chapter 7

Conclusion

This thesis presents the benefits of using point-to-point communication between FPGAs in the MIDG2 application for accelerating computation of electric and magnetic fields using the Discontinuous Galerkin method. The `MIDG2 MPI FPGA` is extended to communicate the shared data between FPGAs using point-to-point communication in two different topologies viz. `within node` and `fully connected`. Two different designs are implemented using the point-to-point communication. The first design (`MIDG2 FPGA IO channels`) extends the OpenCL kernels to include the serial IO external channels to use the point-to-point communication between the FPGAs. Second design (`MIDG2 FPGA Only`) introduces changes in the OpenCL kernel to remove the host application interaction completely.

The topologies are initially evaluated using separate prototypes used to perform tests with synthetic data and data sizes. The evaluation of the topologies shows the superiority of the point-to-point communication over the existing MPI+PCIe communication in terms of the peak bandwidth performance. A maximum bandwidth of 19.84 GB/s is achieved for point-to-point communication in `within node` topology using all the four available channels between two FPGAs compared to 1.94 GB/s for MIDG2+PCIe communication. The point-to-point communication are also found to be more efficient as they are able to utilize the 99.2% of the available bandwidth compared to 65% utilization for the MPI+PCIe communication.

The results of the evaluation with extended `MIDG2 MPI FPGA` also show reduction in the execution time of the application by 30% using `within node` topology and 20% using `fully connected` topology for different mesh sizes ranging from 1k to 200k mesh element. Additional synthetic test with `MIDG2 FPGA IO channels` where higher data size transfers between the FPGAs show a speedup of 2 times for the execution time with a maximum speedup of 3 for 200k with `fully connected` topology. The synthetic tests show that utilizing the point-to-point communication with applications with higher communication demands or higher communication to computation ratio can benefit a lot from the communication with IO channels.

The IO channels design for the MIDG2 FPGA implementation improves the application performance by using the point-to-point communication between the FPGAs. Further improvements are possible by restructuring the partitioning and OpenCL kernel structures easily.

7.1 Future Work

Following updates to the MIDG2 FPGA designs can be performed to optimize the design further and utilize the benefits of the IO channels completely:

- The bottleneck analysis in section 6.2 identified bottlenecks with the data processing efficiency in the `VOLUME` and `SURFACE` kernel. The computation structure in the compute can be updated to increase the computation efficiency of kernels to achieve higher occupancy. This will allow to utilize the complete global memory bandwidth.

- The partitioning scheme of the meshes can be updated to create smaller partitions with higher communication to computation ratio. This will allow to utilize the complete benefit of the point-to-point communication and further reduce the execution time.

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
PDE Partial Differential Equations

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. 3).

[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. 3).

[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. 3, 9).

[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. 3).

[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. 3).

[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. 3).

[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. 3).

[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. 3).

[10] Prof. Dr. Jens Förstner, Yevgen Grynko, and Samer Alhaddad. *Discontinuous Galerkin Method.* Englisch. Paderborn (cit. on p. 10).

[11] 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. 8).

[12] 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, 3, 8–10).

[13] *Intel® FPGA SDK for OpenCL™ Pro Edition Best Practices Guide.* en. Datasheet UG-OCL003. Intel, May 2019, p. 191. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf (cit. on pp. 7, 40, 65).

[14] *Intel® FPGA SDK for OpenCL™ Pro Edition Programming Guide.* en. Datasheet UG-OCL002. Intel, Apr. 2019, p. 212 (cit. on pp. 7, 56).

[15] 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. 3).

[16] 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. 3).

[17] 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, 9–12, 54).

[18] 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).

[19] 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 pp. 3, 27).

[20] 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. 3).

[21] 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. 3).

[22] T. G. Robertazzi. "Toroidal networks". *IEEE Communications Magazine* 26.6 (June 1988), pp. 45–50 (cit. on p. 20).

[23] 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).

[24] 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. 10).

[25] 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. 3).

[26] 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. 3).