Universidade Estadual de Campinas
Instituto de Computação

Ramon Santos Nepomuceno

Enabling OpenMP Task Parallelism on Multi-FPGAs

Habilitando o Paralelismo de Tarefas do OpenMP em Multi-FPGAs

CAMPINAS
2021

Ramon Santos Nepomuceno

Enabling OpenMP Task Parallelism on Multi-FPGAs

Habilitando o Paralelismo de Tarefas do OpenMP em Multi-FPGAs

Tese apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Supervisor/Orientador: Prof. Dr. Guido Costa Souza de Araújo

Este exemplar corresponde à versão final da Tese defendida por Ramon Santos Nepomuceno e orientada pelo Prof. Dr. Guido Costa Souza de Araújo.

CAMPINAS
2021

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

N352e

Nepomuceno, Ramon Santos, 1991-
Enabling OpenMP task parallelism on Multi-FPGAs / Ramon Santos Nepomuceno. – Campinas, SP : [s.n.], 2021.

Orientador: Guido Costa Souza de Araújo.
Tese (doutorado) – Universidade Estadual de Campinas, Instituto de Computação.

1. OpenMP (Programação paralela). 2. Programação paralela (Computação). 3. FPGA (Arranjo de Lógica Programável em Campo). 4. Computação heterogênea. I. Araújo, Guido Costa Souza de, 1962-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Habilitando o paralelismo de tarefas do OpenMP em Multi-FPGAs
Palavras-chave em inglês:
OpenMP (Parallel programming)
Parallel programming (Computer science)
FPGA (Field programmable gate arrays)
Heterogeneous computing
Área de concentração: Ciência da Computação
Titulação: Doutor em Ciência da Computação
Banca examinadora:
Guido Costa Souza de Araújo [Orientador]
Vanderlei Bonato
Nahri Balesdent Moreano
Ricardo dos Santos Ferreira
Ricardo Pannain
Data de defesa: 18-08-2021
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0002-1115-8722
- Currículo Lattes do autor: http://lattes.cnpq.br/0052907638640970

Universidade Estadual de Campinas
Instituto de Computação

Ramon Santos Nepomuceno

Enabling OpenMP Task Parallelism on Multi-FPGAs

Habilitando o Paralelismo de Tarefas do OpenMP em Multi-FPGAs

Banca Examinadora:

- Prof. Dr. Vanderlei Bonato
  ICMC-USP
- Profa. Dra. Nahri Balesdent Moreano
  FACOM-UFMS
- Prof. Dr. Ricardo dos Santos Ferreira
  DPI-UFV
- Prof. Dr. Ricardo Pannain
  IC-UNICAMP
- Prof. Dr. Guido Costa Souza de Araújo (Advisor)
  IC-UNICAMP

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 18 de agosto de 2021

Acknowledgements

The author would like to thank his advisor, Prof. Guido Araujo, for all the support and guidance throughout his Ph.D. studies.

He would also like to thank all his friends and family for all the support given during this long and arduous PhD process. In particular, this work is posthumously dedicated to João Victor, who unfortunately cannot witness this moment.

Finally, the author would like to thank Coordination for the Improvement of Higher Education Personnel - Brazil (CAPES, financing Code 001 grant #88882.329100/2014-01), the Brazilian National Council for Scientific and Technological Development (CNPq, grant #140274/2020-0) and the São Paulo Research Foundation (FAPESP, grant #2018/21761-3) for financing the project.

Resumo

Os aceleradores de hardware baseados em FPGA têm recebido uma crescente atenção nos últimos anos. Um dos principais motivos para isso é que seus recursos reconfiguráveis facilitam a adaptação do acelerador a diferentes tipos de cargas de trabalho. Em alguns casos, os aceleradores baseados em FPGA fornecem maior desempenho computacional e eficiência energética. Foi relatado que o *offload* para FPGA alcança um melhor desempenho quando comparado GPUs e CPUs para algumas aplicações, como Fast Fourier Transform. Esse desempenho pode ser ainda maior se conectarmos vários FPGAs criando um cluster Multi-FPGA. No entanto, programar esses sistemas heterogêneos é um empreendimento desafiador que ainda requer esforços de pesquisa e desenvolvimento para torná-lo realmente simples. O modelo de programação baseado em tarefas OpenMP é uma boa escolha para programar sistemas Multi-FPGA Heterogêneos. Isso advém da capacidade deste modelo de expor um grau mais alto de paralelismo que combina: (a) *offload* de computação para aceleradores; (b) dependências explícitas de dados; e (c) definição das regiões de código para cada dispositivo específico. Com base nisso, o trabalho desta tese estende a infraestrutura LLVM/OpenMP existente, bem como propõe uma plataforma de hardware para ajudar o programador a expressar facilmente o *offload* e o uso de IP-cores disponíveis em um código binário reconfigurável pré-existente (bitstream). Para isto, foi utilizada uma metodologia de co-design, que implementa a arquitetura de hardware e software em paralelo. No lado do software, duas modificações principais foram feitas na implementação do OpenMP: (a) construir um plugin VC709 na biblioteca libomptarget; e (b) modificar o algoritmo de tempo de execução que gerencia o grafo de tarefas. No lado do hardware, foi criada uma infraestrutura baseada no *Target Reference Design* (TRD) da placa Xilinx VC709. As principais contribuições desta tese são: (a) um novo plugin Clang/LLVM que entende placas FPGA como dispositivos OpenMP, e usa a diretiva OpenMP *declare variant* para especificar IPs-cores de hardware; (b) Um mecanismo baseado na dependência de tarefa OpenMP e no modelo de transferência de computação que permite a comunicação transparente de IPs-cores em uma arquitetura Multi-FPGA; (c) Uma arquitetura em hardware, baseada no Target Reference Design da placa VC709, capaz de executar tarefas OpenMP utilizando preexistentes IP-cores; (d) Um modelo de programação baseado em paralelismo de tarefas OpenMP, que torna simples mover dados entre FPGAs, CPUs ou outros dispositivos de aceleração (por exemplo, GPUs), e que permite ao programador usar um único modelo de programação para executar sua aplicação em uma verdadeira arquitetura heterogênea. Resultados experimentais para um conjunto de aplicações de stencil em OpenMP que executaram em uma plataforma Multi-FPGA com 6 placas Xilinx VC709 interconectadas através de links de fibra ótica, mostraram acelerações quase lineares conforme o número de FPGAs e IP-cores por FPGA aumenta.

Abstract

FPGA-based hardware accelerators have received increasing attention in recent years. One of the main reasons for this comes from its reconfiguration capabilities which facilitate the adaptation of the accelerator to distinct types of workloads. In some cases, FPGA-based accelerators provide higher computational performance and energy efficiency. It has been reported that offloading to FPGA achieves better performance when compared with a GPU and CPU for some applications such as Fast Fourier Transform. This performance can be even higher if one connect multiple FPGAs creating a Multi-FPGA cluster. However, programming such heterogeneous systems is a challenging endeavor and still requires research and development efforts to make it really productive. The OpenMP task-based programming model is a good choice for programming such Heterogeneous Multi-FPGA systems. This is indicated by its ability to expose a higher degree of parallelism that combines: (a) computation offloading to accelerators; (b) explicit data dependencies; and (c) definition of the regions of code for each specific device. Based on that, the work of this thesis extends the existing LLVM/OpenMP infrastructure as well as a hardware platform to help the programmer easily express the offloading and use of IP-cores available in an already existing reconfigurable binary code (bitstream). A co-design methodology was used, which implements both the hardware and software architectures in parallel. On the software side, three main modifications to the OpenMP implementation were made: (a) Insert offload information into the construction of the task; (b) a modification in the task graph management mechanism; and (c) the design of the VC709 plugin in the libomptarget library. On the hardware side, an infrastructure based on the Xilinx VC709 board Target Reference Design (TRD) was created. The main contributions of this thesis are: (a) a new Clang/LLVM plugin that understands FPGA boards as OpenMP devices, and uses OpenMP *declare variant* directive to specify hardware IPs-cores; (b) A mechanism based on the OpenMP task dependence and computation offloading model that enables transparent communication of IP-cores in a Multi-FPGA architecture; (c) A hardware architecture, based on the Target Reference Design of the VC709 board, capable of executing OpenMP tasks using pre-existing IP-cores; and (d) A programming model based on OpenMP task parallelism, which makes it simple to move data among FPGAs, CPUs or other acceleration devices (e.g. GPUs), and that allows the programmer to use a single programming model to run its application on a truly heterogeneous architecture. Experimental results for a set of OpenMP stencil applications running on a Multi-FPGA platform consisting of 6 Xilinx VC709 boards interconnected through fiber-optic links, have shown close to linear speedups as the number of FPGAs and IP-cores per FPGA increase.

Contents

Chapter 1

Introduction

With the limits imposed by the power density of semiconductor technology, heterogeneous systems became a design alternative that combines CPUs with domain-specific accelerators to improve power-performance efficiency [80]. A modern heterogeneous system typically combines general-purpose CPUs and GPUs to speedup complex scientific applications [58]. However, for many specialized applications that can benefit from pipelined parallelism (e.g. FFT, Networking), FPGA-based hardware accelerators have shown to produce improved power-performance numbers [50, 74, 90, 87]. Moreover, FPGA's reconfigurability facilitates the adaptation of the accelerator to distinct types of workloads and applications. In order to leverage on this, cloud service companies like Microsoft Azure [33] and Amazon AWS [16] are offering heterogeneous computing nodes with integrated FPGAs.

Given its small external memory bandwidth [43] FPGAs do not perform well for applications that require intense memory accesses. Pipelined FPGA accelerators [65, 53, 24] have been designed to address this problem but such designs are constrained to the boundaries of a single FPGA or are limited by the number of FPGAs that it can handle. By connecting multiple FPGAs, one can design deep pipelines that go beyond the border of one FPGA, thus allowing data to be transferred through high speed links from one FPGA to another without using external memory as temporal storage. Such deep pipelined accelerators can considerably expand the application of FPGAs, thus enabling increasing speedups as more FPGAs are added to the system [93].

Unfortunately, programming such Multi-FPGA architecture is a challenging endeavor that still requires additional research [71, 85]. Synchronizing the accelerators inside the FPGA and seamlessly managing data transfers between them are still significant design challenges that restrict the adoption of such architectures. The work of this thesis addresses this problem by extending the LLVM OpenMP task programming model [9] to Multi-FPGA architectures.

At a higher abstraction level, the programming model proposed in this paper enables the programmer to see the FPGA cluster as a regular OpenMP device where OpenMP tasks are executed by IP-cores accelerators (IPs). In the presented approach, the OpenMP task dependence mechanism transparently coordinates the IP-cores' work to run the application. For example, consider the simple problem of processing the elements of a vector V in a pipelined fashion using four IP-cores (IPO-IP3) programmed in two FPGAs. Each

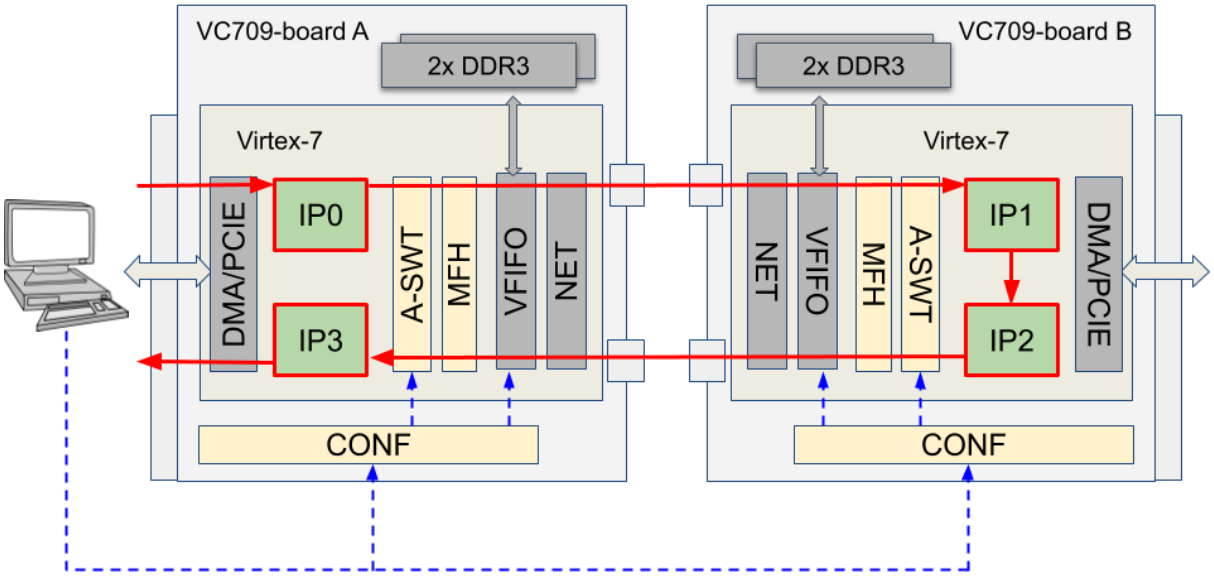


Figure 1.1: An optical-link interconnected Multi-FPGA architecture running an OpenMP pipelined application.

IPi (i = 0-3) performs some specific computation *foo(V,i)*. The Multi-FPGA architecture used in this example is shown Figure 1.1 and contains two VC709 FPGA boards interconnected by two fiber optic links. Each FPGA is programmed with: (a) a module for communication with the PCIE interface (DMA/PCIE); (b) two IP-cores from the set IP0-IP3; (c) a NET module for communication with the optical fibers; (d) a Virtual FIFO module (VFIFO) for communication with memory; (e) a MAC frame handler (MFH) to pack/unpack data; and (f) a packet switch (A-SWT) module capable of moving data among the IP-cores, even if they seat on FPGAs from two distinct boards. More details on the workings and implementation of these modules are given in Chapter 3. As shown in the figure, the vector is initially moved from the host memory (left of Figure 1.1) and then pushed through the IP0-IP3 pipeline, returning the final result into the host memory.

The OpenMP program required to execute this simple application uses just a few code lines (see Listing 1.1). Its simplicity is possible due to the techniques presented in this thesis, which leverages on the OpenMP task dependence and computation offloading abstraction to hide the complexity needed to move the vector across the four IP-cores. The presented techniques extend the OpenMP runtime with an FPGA device plugin which: (a) maps task data to/from the FPGAs; (b) transparently handles the data dependencies between hardware accelerator modules (IP-cores from now on) located in distinct FPGAs; and (c) eases the synchronization of IP-core's execution.

The main contributions of this thesis are summarized below:

- A new Clang/LLVM plugin that understands FPGA boards as OpenMP devices, and uses OpenMP *declare variant* directive to specify hardware IP-cores;
- A mechanism based on the OpenMP task dependence and computation offloading

```
#pragma omp declare variant
  (void do_laplace2d(int*,int,int))  match (device=arch(vc709))
extern void hw_laplace2d(int*,int,int);

int main(){
  float  V[h*w];
  bool deps[N+1];
  #pragma omp parallel
  #pragma omp single
  for (int i = 0; i < N; i++){
    #pragma omp target map(tofrom:V[:(h*w)]) \
    depend(in:deps[i]) depend(out:deps[i+1]) \
    nowait
    {
      do_laplace2d(&V,h,w);
    }
  }
}
```

Listing 1.1: Offloading task computations to FPGA IP-cores.

model that enables transparent communication among IP-cores in a Multi-FPGA architecture;

- A hardware architecture, based on the Target Reference Design of the VC709 board, capable of executing OpenMP tasks using pre-existing IP-cores.
- A programming model based on OpenMP task parallelism, which makes it simple to move data among FPGAs, CPUs or other acceleration devices (e.g. GPUs), and that allows the programmer to use a single programming model to run its application on a truly heterogeneous architecture.

The above contributions have been published in a recognized conference and workshop of the area, as listed below:

1. **R. Nepomuceno**, R. Sterle, and G. Araujo, "Enabling Multi-FPGA Clusters as an OpenMP Acceleration Device," https://jnamaral.github.io/icpp20/slides/Nepomuceno_Enabling.pdf, August 2020, software Stack for Hardware Accelerators Workshop (SSHAW)
2. **R. Nepomuceno**, R. Sterle, G. Valarini, M. Pereira, H. Yviquel, and G. Araujo, "Enabling OpenMP Task Parallelism on Multi-FPGAs," in 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), May 2021, pp. 260-260.

The author of this thesis also contributed to a publication that was a predecessor of this work:

1. C. Ceissler, **R. Nepomuceno**, M. Pereira, and G. Araujo, "Automatic Offloading of Cluster Accelerators," in 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), April 2018, pp. 224–224.

In addition, these works were also presented at national and international events:

1. **R. Nepomuceno**, C. Ceissler and G. Araujo, "Hardcloud: Automatic offloading to cluster accelerators," https://indico.cern.ch/event/683620/contributions/3420618/attachments/1841793/3021386/HardCloud_infieri.pdf, May 2019, iNFIERI: Intelligent signal processing for FrontIEr Research and Industry.

2. C. Ceissler, **R. Nepomuceno**, M. Pereira, and G. Araujo, "Tutorial III - HardCloud / Intel Harp - Automatic Offloading of Cluster Accelerators," http://www2.sbc.org.br/ cradsp/eradsp/2018/programa.html#tut3, April 2018, 9ª Escola Regional de Alto Desempenho (ERAD);

3. C. Ceissler, **R. Nepomuceno**, M. Pereira, and G. Araujo, "HardCloud: The HARP as an OpenMP Acceleration Device" http://wscad.facom.ufms.br/whc.html, October 2017, Workshop on Hybrid Computing 2017 (WSCAD);

4. **R. Nepomuceno**, C. Ceissler and G. Araujo, "Workshop HardCloud" http://wscad.sbc.org.br/2018/programa-wch.html, October 2018, Simpósio de Sistemas Computacionais de Alto Desempenho (WSCAD);

Finally, this work was also submitted to the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems journal (TCAD) and is, at the time of writing this thesis, under review.

The rest of this thesis is organized as follows. Chapter 2 discusses the background necessary to understand the work. Chapter 3 presents, in details, the main contributions of this thesis and the methodology used to implement them. Chapter 4 shows the benchmark used to evaluate the work and the experimental results found. Chapter 5 discusses the related works found in the literature. Finally, the conclusions of the work are presented in Chapter 6.

Chapter 2

Background

This chapter discusses the technical concepts necessary to better understand the work presented in this thesis. Section 2.1 describes the main aspects of the OpenMP programming model with emphasis on task parallelism and the support for device offloading. Section 2.2 presents the main concepts about FPGA acceleration, with a focus on the Xilinx VC709 board and the modules used to assemble the testing platform.

2.1 OpenMP

Although computers are always getting faster, the demand for computing power is also always increasing. A more powerful machine leads to new kinds of applications, which in turn fuel the demand for yet more powerful systems. Engineers develop several techniques to support this demand for performance. Among them is the use of multiple components and functional units that may be able to operate simultaneously on specific tasks, such as adding two integer numbers or determining whether a value is greater than zero, for example. This very low level parallelism approach is often referred to as Instruction-Level Parallelism (ILP) [21, 86]. To leverage on this kind of parallelism, compiler writers developed techniques to better organize the instructions to efficiently utilize ILP. Modern compilers put considerable effort into this kind of optimization. Unfortunately, there is a limited payoff for extending the hardware support [94, 66].

Back in the 1980s, several vendors produced computers that exploited another kind of architectural parallelism. They built machines consisting of multiple processors [35] with a common shared memory. These multiprocessor machines could work on several tasks at once, by simply executing them on different processors. As a result, they became popular in the server market, where they have remained important ever since.

Once the vendors had the technology to build moderately priced parallel machines, they needed to ensure that their computing power could be exploited by individual applications. This is where things got complicated. Compilers had always been responsible for adapting a program to make the best use of a machine's internal parallelism. Unfortunately, it is very hard for them to do so for a computer with multiple processors or cores. The reason is that compilers must then identify independent blocks of instructions that can be executed in parallel. Techniques to extract such instruction blocks from a se-

```
1
2 #pragma omp directive-name [[,] clause[ [,] clause] ... ] new-line
3                                                                    .
```

Listing 2.1: OpenMP directives for C/C++.

quential program do exist; and, for simple programs, or specific cases (e.g. vectorization) it may be worthwhile trying out compiler's automatic parallelization features. However, the compiler often does not have enough information to decide whether it is possible to split up a particular program in its parallel components. It also cannot make large-scale changes to code, such as replacing an algorithm that is not suitable for parallelization. Thus, most of the time the compiler needs some sort of hints from the programmer [23, 81].

For this reason, vendors of parallel machines in the 1980s [36] provided special notation to specify how the work of a program was to be divided among the individual processors, as well as to enforce an ordering of accesses by different threads to shared data. The notation mainly took the form of special instructions, or directives, that could be added to programs written in sequential languages e.g. Fortran. The compiler used this information to create the actual code for execution by each processor. Although this strategy worked, it had the problem that a program written for one vendor did not necessarily execute on another one.

During the latter half of the 1990s, a group of vendors joined forces to resolve this problem by providing a common way for programming a broad range of parallel machines. This group is known as OpenMP Architecture Review Board (ARB) [1], and they were responsible for creating an Application Programming Interface (API) to enable a portable shared memory parallel programming model, called OpenMP [45]. The first version, consisting of a set of directives that could be used with Fortran, was introduced to the public in late 1997. OpenMP compilers began to appear shortly thereafter. Since that time, bindings for C and C++ have been introduced, and the set of features has been extended. OpenMP ready compilers are now available for almost all computers. The number of vendors involved in maintaining and further developing OpenMP features has grown. Today, almost all the major computer manufacturers, major compiler companies, several government laboratories, and groups of researchers belong to the ARB.

OpenMP can be defined as a set of compiler directives, runtime library routines, and environment variables to specify parallelism in Fortran and C/C++ programs. An OpenMP directive is a specially formatted *pragma* that applies to the sequential code block/statement immediately following it in the program. These directives let the user tell the compiler which instructions to execute in parallel and how to distribute them among the threads that will run the code. An OpenMP directive is an instruction in a special format that is understood by OpenMP-aware compilers only. They look like a comment to a regular Fortran compiler or a pragma to a C/C++ compiler, so that the program may run just as it did beforehand if a compiler is not OpenMP-aware.

Listing 2.1 shows the specification of an OpenMP directive for C/C++. Each directive starts with **#pragma omp**. The *directive-name* is the name of the directive and can possibly have any directive-level arguments enclosed in parentheses. The clauses are

```
1 #include <omp.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(int argc, char* argv[]){
6   #pragma omp parallel
7   {
8     printf("thread number = %d\n", omp_get_thread_num());
9   }
10  return 0;
11 }
```

Listing 2.2: OpenMP Hello World.

modifiers that can be applied to certain directives.

The first step in creating an OpenMP program from a sequential one is to identify the parallelism it contains. Basically, this means finding blocks of codes that can be executed concurrently by different processors. Sometimes, the developer must reorganize its program to obtain independent instruction sequences. It may even be necessary to replace an algorithm with an alternative one that accomplishes the same task but offers more exploitable parallelism.

The second step in creating an OpenMP program is to express, using OpenMP, the parallelism that has been identified. One of the advantages of using OpenMP is that it can incrementally create a parallel program from an existing sequential code. The developer can insert directives into a portion of the program and leave the rest in its sequential form. Once the resulting program version has been successfully compiled and tested, another portion of the code can be parallelized. These are characteristics that encouraged the adoption of OpenMP as the programming model for the development of the work of this thesis.

The code in Listing 2.2 shows a trivial example of a program parallelized using OpenMP. The library included in line 1 (#include <omp.h>) has the signatures of OpenMP functions, such as the function *omp_get_thread_num()* called in line 8. This function returns an identifier number of the thread that is running. The #**pragma omp parallel** directive on line 6, is the directive responsible for creating the threads that execute in parallel. The number of threads to be created can be defined by calling the *omp_set_num_threads* function or via the OMP_NUM_THREADS environment variable. So, this simple example shows the use of the three main components of an OpenMP program: compiler directives, runtime library routines, and environment variables.

OpenMP supports the so-called fork-join programming model [46], which is illustrated in Figure 2.1. The idea of this approach is to start the program as a single thread of execution, just like a sequential program. The thread that executes this code is referred to as the initial thread and it executes the serial region. Whenever an OpenMP parallel construct is encountered by a thread, it creates a team of threads (this is the fork) and together they execute the parallel region. At the end of the parallel region, only the original (initial) thread continues, starting another serial region; all others terminate (this is the join).

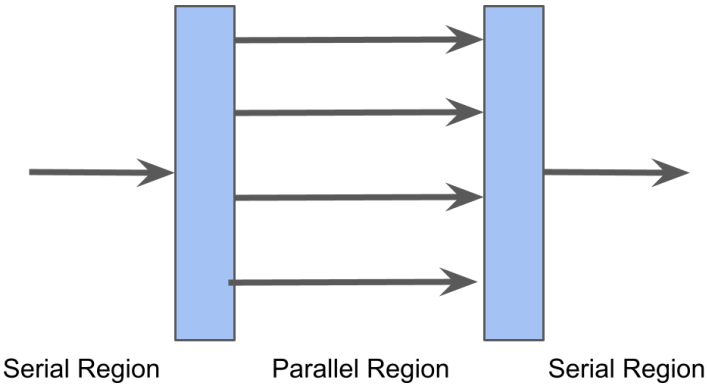


Figure 2.1: Fork-Join Programming Model.

OpenMP expects the programmer to give a high-level specification of the parallelism in the program and the method for exploiting that parallelism. Thus, it provides notation for indicating the regions of an OpenMP program that should be executed in parallel. It also enables the provision of additional information on how this is to be accomplished. The job of the OpenMP implementation is to deal with the low-level details of actually creating independent threads to execute the code, and to assign work to them according to the strategy specified by the programmer.

In addition to the classic fork-join model, where all threads in the parallel region perform the same code, the OpenMP model also allows expressing parallelism using the concept of *tasks*, which is presented in the next section.

2.1.1 OpenMP Task Directive

Tasks were first introduced to OpenMP in version 3.0 [8]. Before that, the distribution of work between tasks was mostly based on directives aimed at array-based applications, such as the parallelization of *for* loops using the directive *parallel for*. There was no standardized mechanism to express and exploit unstructured parallelism efficiently and elegantly. The main idea behind tasks is to allow the programmer to expose a more complex degree of parallelism by using the directive *task*. This was combined with the *depend* clause, that makes data dependencies explicit to facilitate parallelization of applications where units of work are generated dynamically. In the OpenMP specification, a task is defined as a specific instance of executable code and its data environment (inputs to be used and outputs to be generated) [11].

Initially, tasks were only implicit in OpenMP. An implicit task is a task generated when a parallel construct is encountered during execution. A parallel directive builds implicit tasks, one per thread, and all tasks would execute the same code and synchronize using a barrier once all tasks are completed at the end of the parallel region. All implicit tasks, generated when a parallel construct is encountered, are guaranteed to be complete when the control thread exits the implicit barrier.

With the creation of the task parallelism model, OpenMP started to allow programmers to explicitly create tasks, thus providing support for different executions of the code and enabling more flexibility since now the threads can execute multiple existing tasks.

```
#pragma omp task [clause[ [,] clause] ... ] new-line
/* structured-block */
```

Listing 2.3: OpenMP Task Directives for C/C++.

```
/* some code */

#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task depend(out: A)
    {
      A = foo();
    }
    for(int i=0; i < 2 ; i++){
      #pragma omp task depend(in: A) depend(out: B[i])
      {
        B[i] = bar(A);
      }
    }
    for(int i=0; i < 4 ; i++){
      #pragma omp task depend(in: B[i/2])
      {
        fun(B[i/2]);
      }
    }
  }
}

/* some code */
```

Listing 2.4: OpenMP task example

An explicit task is specified using the *task* directive. Listing 2.3 shows the formalization of an OpenMP task directive for C/C++. There are several clauses that can be associated with the task directive. However, this thesis focuses on the *depend* clause that is used to express data dependencies.

The *task* directive defines the code associated with the task and its data environment. Whenever a thread encounters this directive, a new task is generated. According to the dependencies of this task, a thread can execute it immediately or defer its execution until a later time. If the execution is deferred, then the task is placed in a ready queue of tasks that is associated with the current parallel region. The threads in the current team will take tasks out of the queue and execute them until it is empty. A thread that executes a task may be different from the thread that originally encountered it. All explicit tasks generated within a parallel region are guaranteed to be complete on exit from the next implicit or explicit barrier within the parallel region.

An example of a program that uses OpenMP tasks is shown in Listing 2.4. Figure 2.3 illustrates, in a simplified way, how task management is done. The actual implementation is more complicated, but the figure helps to understand the overall idea.

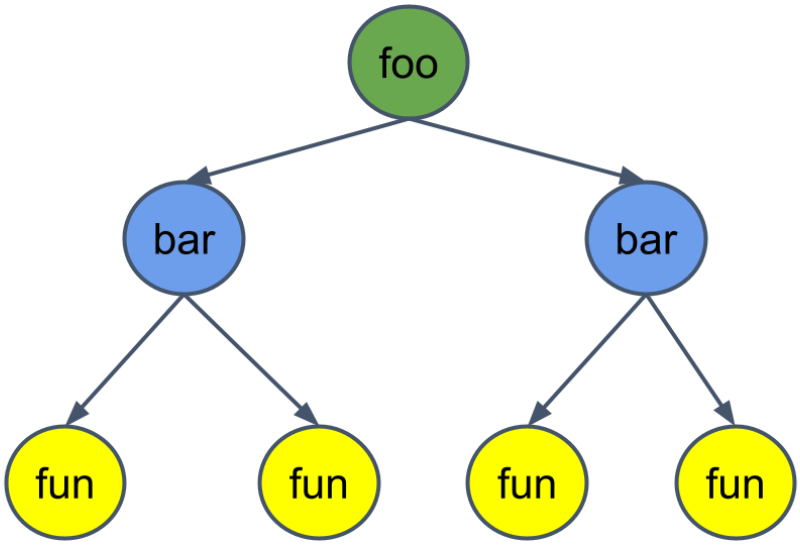


Figure 2.2: OpenMP task graph.

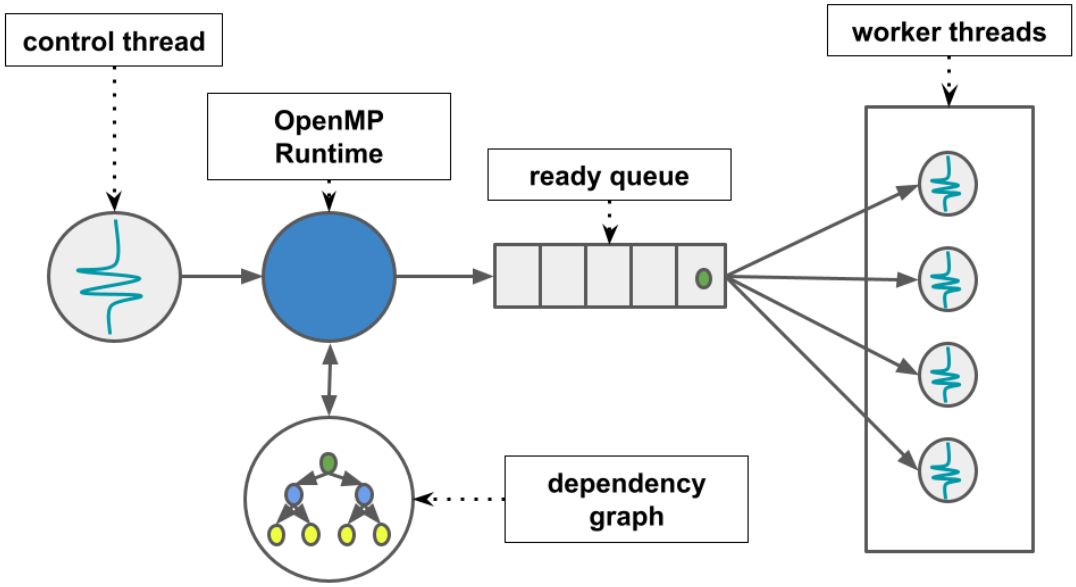


Figure 2.3: LLVM/OpenMP Runtime Implementation.

First, as already said, the program creates a pool of *worker threads* (Figure 2.3). This is done using the *#pragma omp parallel* directive (Listing 2.4 line 3). It is with this set of threads that an OpenMP program runs in parallel.

Next, the *#pragma omp single* directive (Listing 2.4 line 5) is used to select one of the *worker threads*. This selected thread is responsible for creating the tasks, and will be called *control thread* (Figure 2.3) from now on.

The *#pragma omp task* directive (Listing 2.4 lines 7, 12 e 18) is used to effectively create *tasks*, while the *depend* clause specifies the input and output dependencies of a given task. The *control thread* of Listing 2.4 creates seven tasks. The dependencies between these tasks form a dependency graph, which is illustrated in Figure 2.2.

The OpenMP runtime manages the execution of tasks (Figure 2.3). Whenever the

```
1
2 #pragma omp target [clause[ [,] clause] ... ] new-line
3 /* structured-block */
```

Listing 2.5: OpenMP Target Directives for C/C++.

dependencies of a given *task* are satisfied, that task goes to a *ready queue*. Once queued, it can be executed by any *worker thread* (Figure 2.3).

Another OpenMP directive used in this work is the *target* directive. The following section explains how it works.

2.1.2 OpenMP Target Directive

Heterogeneous computing is here to stay [97]. However, much effort will still be required to create means to facilitate the programming of such systems. OpenMP tackles this problem by supporting computing offload to acceleration devices through the *target* directive. This directive offers a path to more portable device accelerated software. One of the goals of this standard is to minimize the need for programs to contain device vendor-specific statements, making the codes portable across different architectures.

OpenMP device offloading is a host-centric model with one host device and possible multiple target devices attached to it. A device is a logical execution engine with local storage. The device data environment is an environment associated with a target data or target region. Target constructs control how data and code are offloaded to a device and the data is mapped from a host data environment to a device data environment.

The target region is the basic offloading construct in OpenMP. A target region defines a section of a program that will be offloaded. The OpenMP program starts executing on the host and when a target region is encountered, the code it contains is executed on a device.

Listing 2.5 shows the formalization of an OpenMP target directive for C/C++. The directive starts with the *#pragma omp target* construct which is followed by some clauses. Four are more relevant to this work: *device*, *map*, *depend* and *nowait* clauses. The *device* clause receives as an argument an integer that specifies the device to which the offload will be performed. If this clause is not used, a default device is selected. The *map* clause specifies the action that the host must execute with a given data. Some of the actions available are *to*, *from*, *tofrom* or *alloc*. The first three of them specify if the data should move to/from the device and the last one only allocates the space into the device memory. The *depend* clause has the effect of turning the offload region into a task as if the target directive were a *task* directive. The clause *nowait* has the effect of making the host continue its execution without waiting for the target region to be finalized, the synchronization will occur in the next barrier. More details in the *depend* an *nowait* clauses are given in Section 2.1.3.

An illustrative example of a program that uses OpenMP *target* directive to offload to a device is shown in Listing 2.6. In the example, the code under the scope of the target directive (line 2) will run on an accelerator device. The *map* clause states which data will

```
 /*some code*/
#pragma omp target map(to: X[:N]) map(from: Y[:N])
{
  for(int i = 0; i < N; i++){
    Y[i] = X[i];
  }
}
/*some code*/
```

Listing 2.6: OpenMP target example

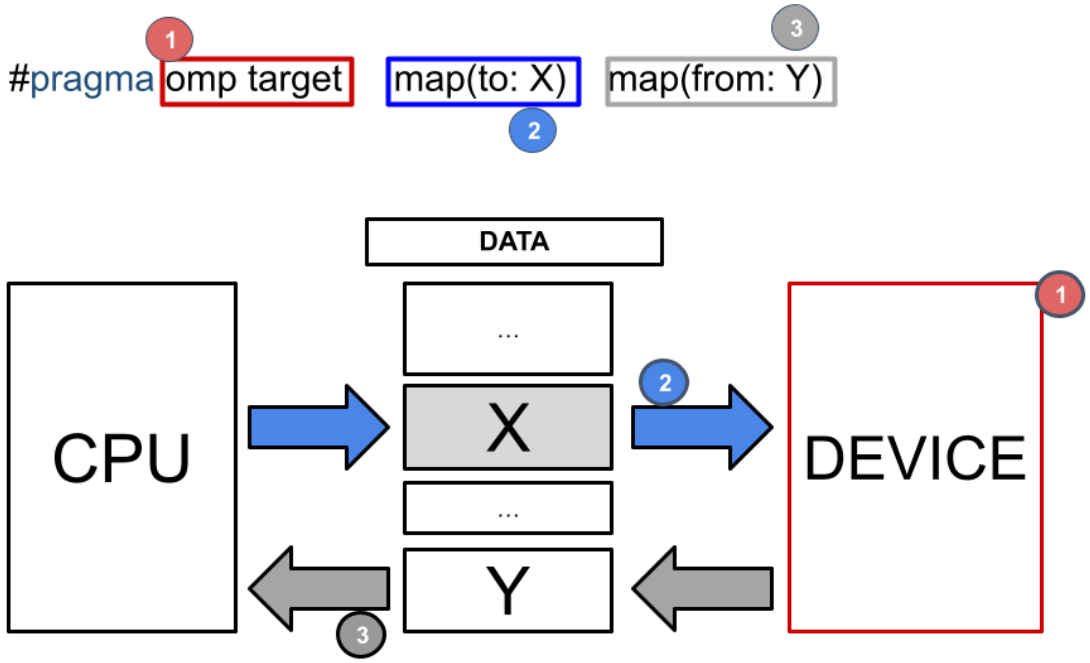


Figure 2.4: OpenMP offloading.

be moved to (X) and from (Y) the device.

Figure 2.4 illustrates, in a simplified way, what the code execution would look like. First, the OpenMP runtime sends the X buffer to the device. It then starts code execution. And finally, it retrieves the result that was stored into buffer Y.

In the OpenMP implementation for the Clang/LLVM compiler [73], offloading is performed by a library called *libomptarget* [26]. This library provides an agnostic offloading mechanism, in which to add a new device to the list of devices that the OpenMP runtime supports, one just needs to create and add a new plugin to the library. Figure 2.5 shows how the *libomptarget* library interacts with the system.

First, Clang generates a *fat binary* with the host and targets appropriate representation, (e.g. ELF, PE32+, and Mach-O). At the moment that the program reaches a target directive, the library checks if the plugin is compatible with the target binary, then maps the data environment and executes the computation on the device that was specified.

The OpenMP specification enables the user to create dependencies among tasks on distinct devices, even without a *task* directive. This is done using the *target* directive along with the *depend* clause. The next section explains how this mechanism works.

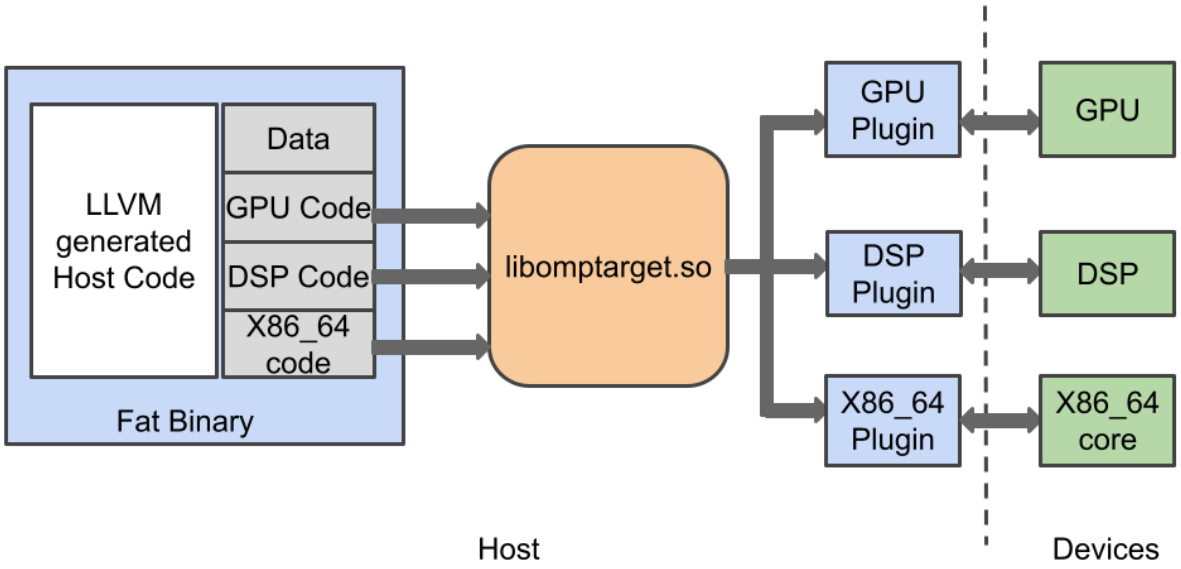


Figure 2.5: Libomptarget schematic.

2.1.3 OpenMP Target Depend Directive

As already said, the *target* directive also accepts the *depend* clause. That way, even without using the *task* directive, a programmer can create a task that runs on a device. The program shown in Listing 2.7 rewrites the program of Listing 2.4 so that tasks are now executed on an accelerator device.

Notice that the clause *nowait* also appears in the code of Listing 2.7. This clause is necessary because, by default, the target directive is blocking. In other words, the nowait clause allows the control thread to create all seven tasks without waiting for the previous ones to complete.

The current LLVM/OpenMP implementation of task offloading does not support devices like Multi-FPGA. For this reason, the work of this thesis presents an OpenMP plugin implementation for Multi-FPGA devices. More details on the implementation are given in Chapter 3.

The adoption of OpenMP in the work of this thesis can be attributed to a number of factors. One is its strong emphasis on structured parallel programming. Another is that OpenMP is comparatively simple to use, since the burden of working out the details of the parallel program is up to the compiler. It has the major advantage of being widely adopted, so that an OpenMP application will run on many different platforms. But above all, OpenMP is timely. The vendors behind OpenMP collectively deliver a large fraction of the computer machines in use today. Their involvement with this standard ensures its continued applicability to their architectures. Furthermore, the ARB continues to work to ensure that OpenMP remains relevant as computer technology evolves. OpenMP is under cautious, but active, development; and features continue to be proposed for inclusion into the application programming interface. Applications live vastly longer than

```
/* some code */

#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp target nowait           \\
    map(from: A)                        \\
    depend(out: A)
    {
      A = foo();
    }
    for(int i=0; i < 2 ; i++){
      #pragma omp target nowait         \\
      map(to: A) map(from: B[i])        \\
      depend(in: A) depend(out: B[i])
      {
        B[i] = bar(A);
      }
    }
    for(int i=0; i < 4 ; i++){
      #pragma omp target nowait         \\
      map(to: B[i/2])                   \\
      depend(in: B[i/2])
      {
        fun(B[i/2]);
      }
    }
  }
}

/* some code */
```

Listing 2.7: Tasks in Devices

computer architectures and hardware technologies; and, in general, application developers are careful to use programming languages that they believe will be supported for many years to come.

The next section explains important concepts regarding the hardware part of the work of this thesis, that is, reconfigurable architectures, more precisely the FPGA VC709 board from Xilinx.

2.2 The Hardware Platform

In general, computers can be classified according to their purpose, being either *General Purpose* or *Specific Application Purpose* [31]. A General-Purpose computer is a single silicon chip, called microprocessor or GPP, that could be programmed to solve many computing tasks. This means that many applications could share commodity economics for the production of a single integrated circuit (IC). An Application-Specific Integrated Circuit (ASIC) is an IC specifically designed to provide unique functions. ASIC chips can replace general-purpose commercial logic chips, and integrate several functions or logic control blocks into one single chip. Although the ASIC has the advantages of high performance and low power, its fixed resource and algorithm architecture result in drawbacks such as high fabrication cost and poor flexibility. As a tradeoff between the two extreme characteristics of GPP and ASIC, there is the *Reconfigurable Computing* that combine the advantages of both [52].

2.2.1 Reconfigurable Computing

The work by Compton at al in [42], presents reconfigurable computing as intended to fill the gap between hardware and software. This gap is filled by achieving potentially much higher performance than software while maintaining a higher level of flexibility than hardware. Reconfigurable computing can be used by implementing all of the application functionalities in hardware. In this case, the hardware implemented on the reconfigurable elements covers all the data path from the inputs to the outputs of the application. The advantage in doing this is that the hardware is easily replaceable by downloading an appropriate configuration file onto the chip, rather than having the circuit physically replaced. Therefore, it can be concluded that reconfigurable computing is a trade-off between general-purpose computing and application-specific computing because it tries to achieve a balance among performance, cost, power, flexibility, and design effort. Reconfigurable computing has enhanced the performance of applications in a large variety of domains, including embedded systems [57], SoCs [79], digital signal processing (DSP) [91], image processing [61, 84, 92], network security [37], bioinformatics [39, 47], supercomputing [51, 72, 49, 56, 63], Boolean SATisfiability (SAT) [82, 20], spacecrafts [48], and military applications [76]. It can be said that reconfigurable computing will widely, pervasively, and gradually impact human lives.

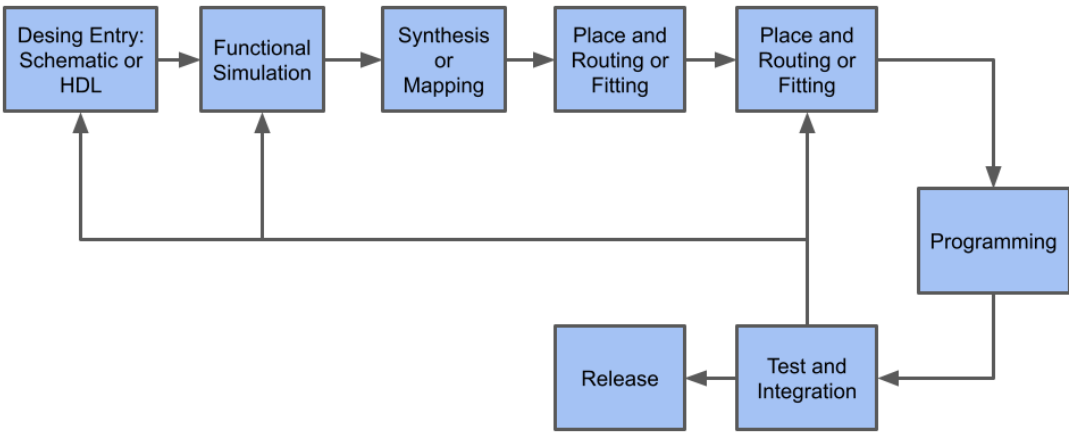


Figure 2.6: FPGA Development Process Design Flow.

2.2.2 The FPGAs

Although *Reconfigurable Computing* was first proposed in the 1960s by Estrin [52], it was only in the 1980s that this type of computing began to gain industrial acceptance with the advent of the first Field Programmable Gate Arrays (FPGAs) [32].

FPGAs are digital integrated circuits that contain configurable (programmable) blocks of logic along with configurable interconnects between these blocks. Design engineers can configure (program) such devices to perform a tremendous variety of tasks. The "field programmable" portion of the FPGA's name refers to the fact that its programming takes place "in the field" (as opposed to devices whose internal functionality is hard-wired by the manufacturer). This may mean that FPGAs are configured in the laboratory, or it may refer to modifying the function of a device resident in an electronic system that has already been deployed in the outside world [77].

Configuring an FPGA means changing its functionality to support a new application, and it is equal to having some new piece of hardware, mapped on the FPGA chip to implement a completely new functionality. In other words, FPGAs make it possible to have custom-designed high-density hardware in an electronic circuit, with the added bonus of having the possibility of changing it whenever there is the need, even while the whole application is still running.

A Hardware Design Language (HDL) such as VHDL or Verilog is used to describe the functionality to be implemented on the device. The design software of the device manufacturer, a.k.a. Electronic Design Automation (EDA) tools, translates the description of the hardware into a configuration file for the device that can be downloaded on it.

EDA tools for FPGAs are analogous to those employed in ASIC chip design, in the sense that they convert a hardware specification into an actual netlist that can be synthesized, placed, and routed on an actual piece of hardware such as an ASIC or an FPGA. The tools use a process flow like the one depicted in Figure 2.6. The design begins with

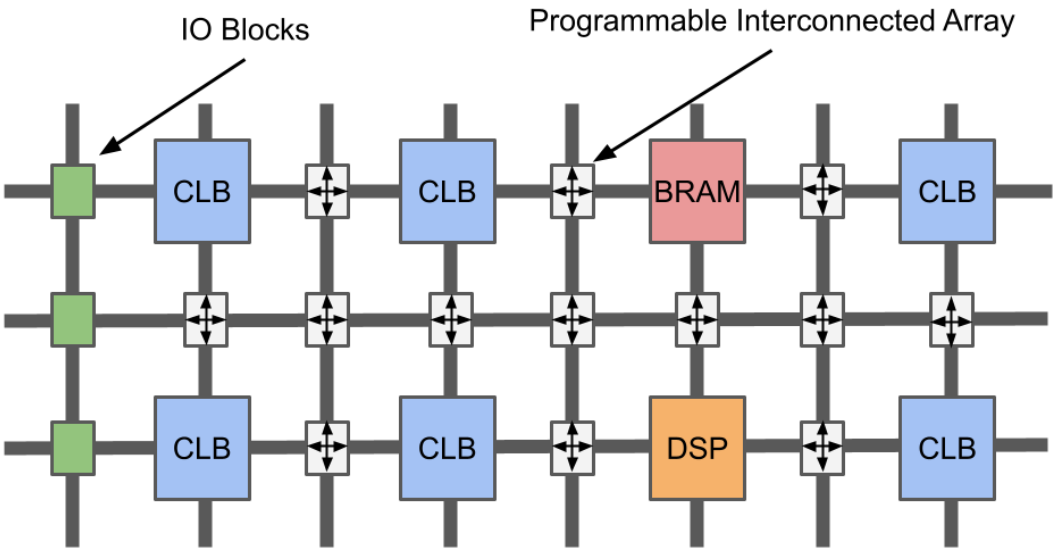


Figure 2.7: Simplified FPGA.

design entry, continues to simulation to verify the logic is implemented as expected. And then the tool performs synthesis, also called mapping, to map the logic to the device architecture. Next, the tool creates the interconnection between the cells by place and routing or fitting the design. It is good practice to run another simulation after the fitting is completed. If the design looks good, the next step is to use the tool to create a programming file, which is then downloaded into the FPGA device for testing. In the programming model presented in this thesis, all these implementation steps are done for the IP-cores, in a separate flow from the software development. This way, the hardware complexity is abstracted from the developer of the final software, easing the design of the whole system.

The flexibility of having custom, adaptable hardware in an application is the factor that has determined the popularity of FPGA devices in a broad range of fields. If the FPGA is paired with a general purpose processor, for example, the most demanding sections of the software can be translated into hardware cores that accelerate program execution, yielding notable speedups in the overall execution, especially when software sections executed serially on the processor can be translated into hardware that can exploit the parallelism of the algorithm.

The FPGA that was used for the development of the project presented in this thesis was one produced by Xilinx, namely Virtex-7. For this reason, the architectural details discussed focus on Xilinx FPGAs, but the main concepts can be applied to the FPGAs of the most diverse companies.

Figure 2.7 illustrates an FPGA containing CLBs, IO Blocks, BRAM, DSP and Programable Interconnected Array (other possible resources have been omitted). The three main building blocks of a Xilinx FPGA are Configurable Logic Blocks (CLB), IO Blocks, and Programmable Interconnected Array. These FPGAs also include memory elements composed of simple flip-flops or more complete blocks of memories. Besides the indispensable programmable logic blocks, modern FPGAs also include other blocks, which are

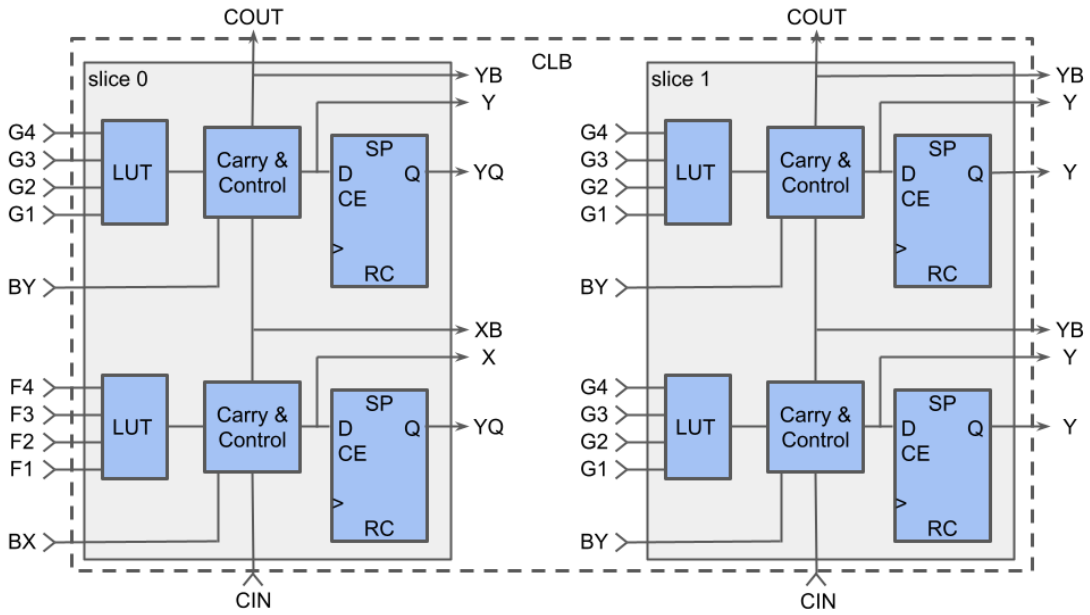


Figure 2.8: CLB schematic.

helpful for the development of large and/or complex designs, such as DSP blocks.

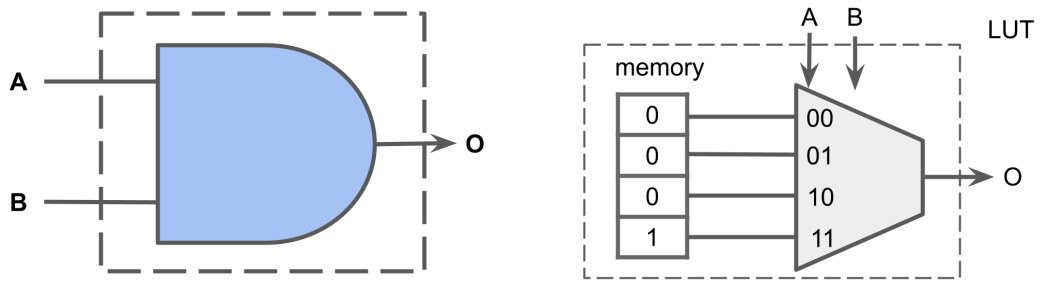
2.2.2.1 FPGA Architecture

Configurable Logic Blocks (CLB) are the main components of a Xilinx FPGA. These elements are the ones that are going to be configured to implement the logic. They can be used to implement either combinational or sequential logic. A CLB can be seen as a prefabric construction that can be customized according to the user's needs. The same starting building block can be updated to meet new requirements. As show in Figure 2.8, in Xilinx FPGAs a single CLB is a hierarchical structure composed of a set of *slices* which, in turn, are composed of a set of *look-up tables*.

The number of slices can vary according to the device, but in general a CLB contains a set of two slices, which is the case for the Virtex-E, V5 or the 7 Series, while four slices were present in the Virtex II Pro and Virtex 4 devices. A slice, in turn, contains two *look-up tables* and the necessary interconnect hardware. The look-up tables are elements that can be used to implement, in general, 4-input, 1 output functions. However, FPGAs like the Xilinx 7 Series are composed of look-up tables that can be used to implement functions characterized by 6-inputs and 2 outputs.

In the Figure 2.8 we can see a 2-slice Virtex-E CLB. This CLB is composed of two slices, each of them containing 2 lookup tables. Therefore, in the end, the CLB is composed of 4 look-up tables. Look-up tables (LUT) can implement an arbitrary logic function according to their configuration. Ultimately, a look-up table, in an FPGA, is nothing more than a memory containing memory cells to implement a small logic function.

Figure 2.9 illustrates how a LUT works. Take as an example the *and* combinational circuit shown in Figure 2.9a, which receives bits A and B as input and produces bit O as output. Figure 2.9b shows a LUT composed of a 4-position memory, referring to the possible output values of a circuit with two input bits, and a multiplexer of two



(a) An *and* combinational circuit. (b) A lut configurated to an *and* circuit.

Figure 2.9: A simple configuration example of a LUT.

values, referring to those same bits. Note that for the values of A equal to '1' and B equal to '1', the multiplexer selects the memory position whose value is equal to '1', and for all other combinations of values of A and B, the multiplexer selects a position in the memory with value 0. What results in an 'and' operation with values A and B. Within this context, configuring a LUT means properly store the necessary sequence of zeros and ones in the 4 memory cells, according to the desired function. Each line is then connected to a multiplexer which is used to read the output of the function we are looking for by selecting/connecting the desired memory cell, via the proper configuration of the multiplexer, set by reading the A and B inputs, to the output signal of the look-up table.

Around the LUT there is interconnect logic (not shown in the figure) that routes signals to and from the LUT, implemented using standard logic gates, multiplexers, and latches. Therefore, during the configuration process of an FPGA, the memory inside the look-up tables is written to implement a required function, and the logic around it is configured to route the signals correctly in order to build a more complex system around this basic building block.

The *input-output blocks* (IOBs) have the function of interconnecting the signals of the internal logic to an output pin of the FPGA package. There is one and only one IOB for every I/O pin of the chip package. The IOBs have their own configuration memory, storing the voltage standards to which the pin must comply and configuring the direction of the communication on it, making it possible to establish mono-directional links in either way or also bidirectional ones. The Input-Output Blocks can be seen as a standard Input-Output interfaces such as a mic input jack, audio output, etc.

The Programmable Interconnect Array within an FPGA allows the arbitrary connection of CLBs, IOBs, BRAM, DSPs and any other resources. The main modes of interconnections are direct and segmented. Direct Interconnection is made of groups of connections that cross the device in all its dimensions. Logic blocks put data on the most suitable channel according to data destination. This implementation usually includes some additional short-range connections that link nearby blocks. Segmented Interconnection is based on lines that can be interconnected using Programmable Switch Boxes. Also in this kind of interconnection there are lines that cross the entire device, in order to maximize the speed of communication and limit signal skew.

The BRAM is a dual-port RAM module instantiated into the FPGA fabric to provide on-chip storage for a relatively large set of data. The two types of BRAM memories

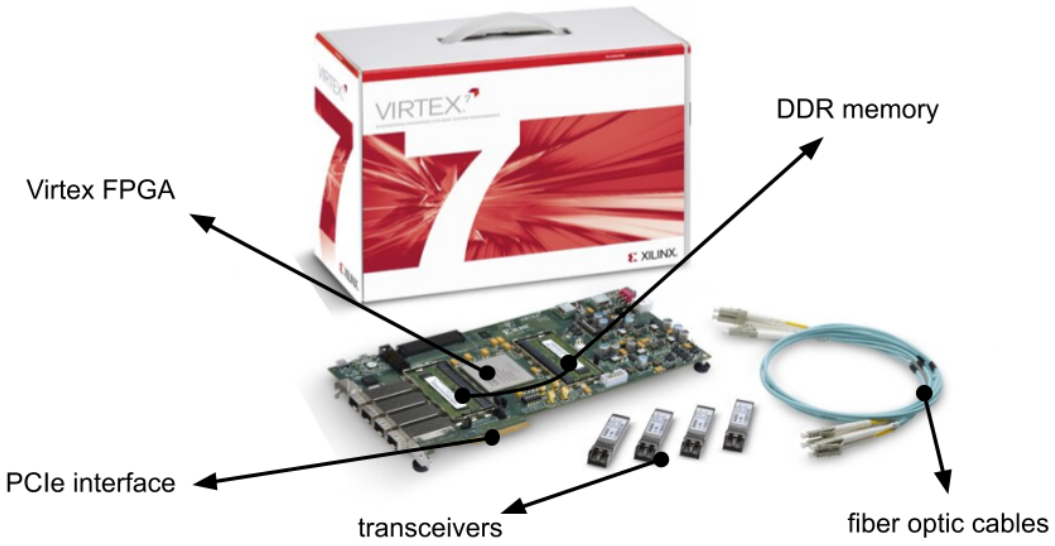


Figure 2.10: VC709 Board componentes. Fonte: [14]

.

available in a device can hold either 18k or 36k bits, and the available amount of these memories is device specific. The dual-port nature of these memories allows for parallel, same-clock-cycle access to different locations.

Digital signal Processing (DSP) is often required in applications involving audio or video, among others. Such processing (FIR/IIR filtering, FFT, DCT, etc.) is accomplished by three basic elements: multipliers, accumulators (adders), and registers. To make this type of application simpler to implement (route) and also faster (less routing delays), special DSP blocks are included in modern FPGAs, normally containing large parallel multipliers, MAC (multiply-and-accumulate) circuits, and shift registers

The Virtex 5 FPGA also contains DSP blocks (called DSP48E Slices). Each block includes a 25 $\times$ 18 multiplier, plus MACs, registers, and several operating modes. The total number of such blocks varies from 32 (in the smallest device) to 192 (in the largest device), with a maximum frequency of 550 MHz.

The next section describes, in more details, the FPGA board used in the design of this thesis, called VC709 Board.

2.2.3 The VC709 Board

The Xilinx VC709 Connectivity Kit (Figure 2.10) provides a hardware environment for developing and evaluating designs targeting the Virtex-7 FPGA. The VC709 board provides features common to many embedded processing systems, including dual DDR3 memories, an 8-lane PCI Express interface [28], and small SFP connectors for fiber optical (or Ethernet). In addition to the board's physical components, the kit also comes with a Target Reference Design (TRD) featuring a PCI Express IP, a DMA IP, a Network Module, and a Virtual FIFO memory controller interfacing to DDR3 memory. Figure 2.11 shows the

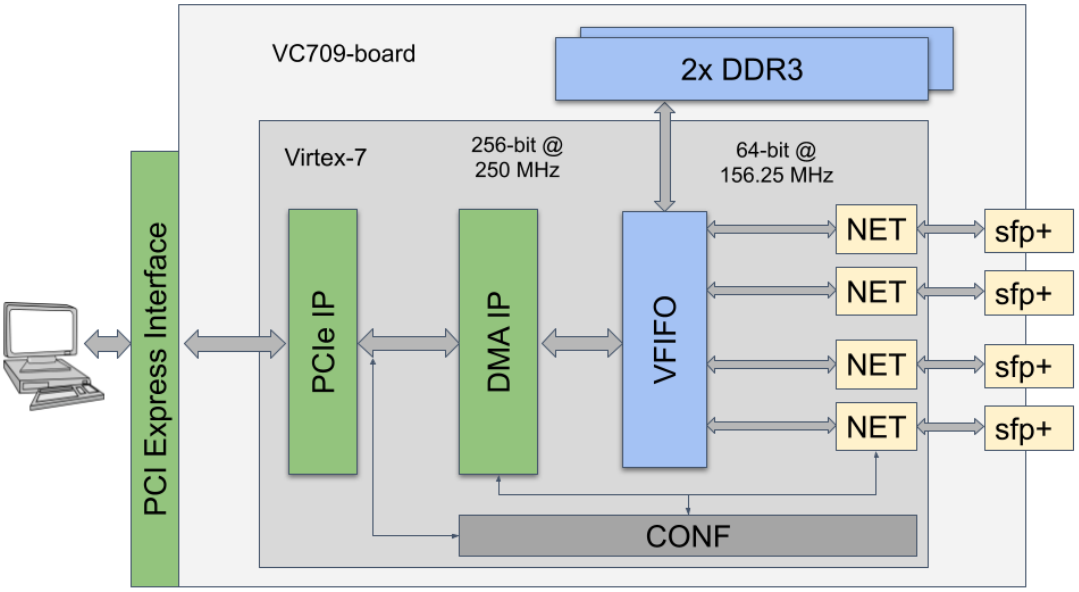


Figure 2.11: VC709 Target Reference Design.

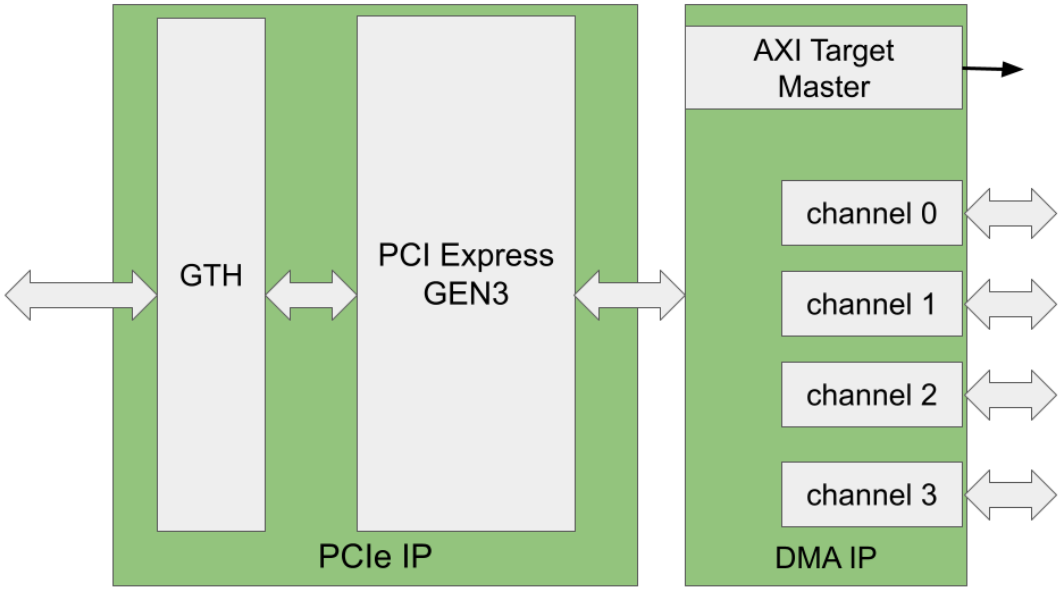


Figure 2.12: PCIe Schematic.

schematic of the main board's components.

This kit was selected to explore the ideas discussed in this thesis due to its reduced cost and the fact that the TRD has components ready for inter-FPGAs communication. Each one of the TRD components is explained in more detail below.

PCIE and DMA: the PCI Express IP (Figure 2.11 and Figure 2.12) provides a wrapper around the integrated block in the FPGA. The integrated block is compliant with the PCIe v3.0 specification. It supports x1, x2, x4, x8 lane widths operating at 2.5 Gb/s (Gen1), 5 Gb/s (Gen2), or 8 Gb/s (Gen3) line rate per direction. The wrapper combines the Virtex-7 XT FPGA Integrated Block for PCIe with transceivers, clocking, and reset logic to provide for the user interface an industry standard AXI4-Stream interface.

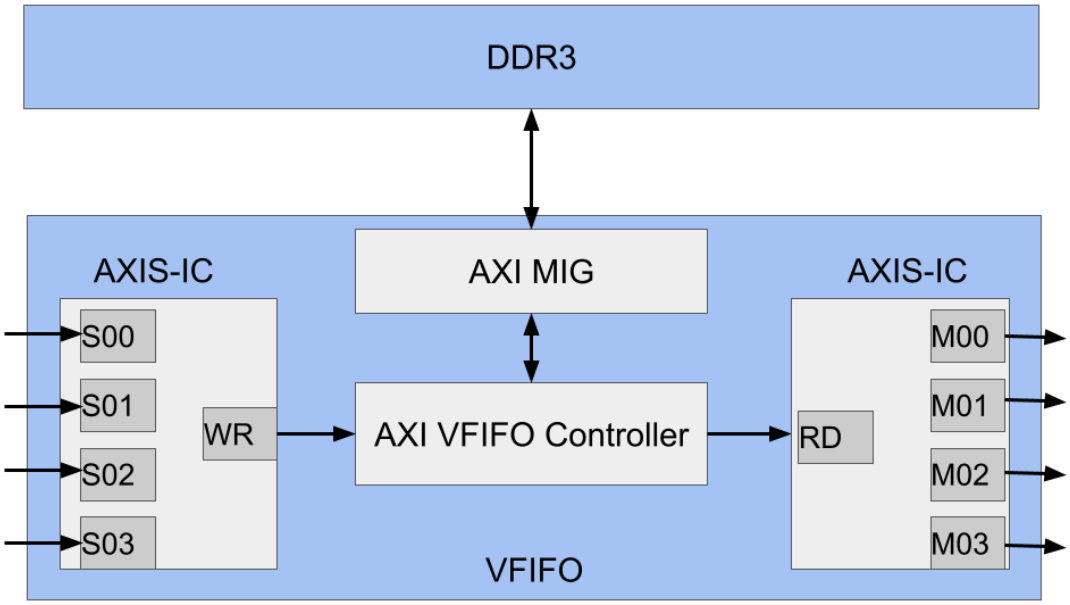


Figure 2.13: VFIFO Schematic.

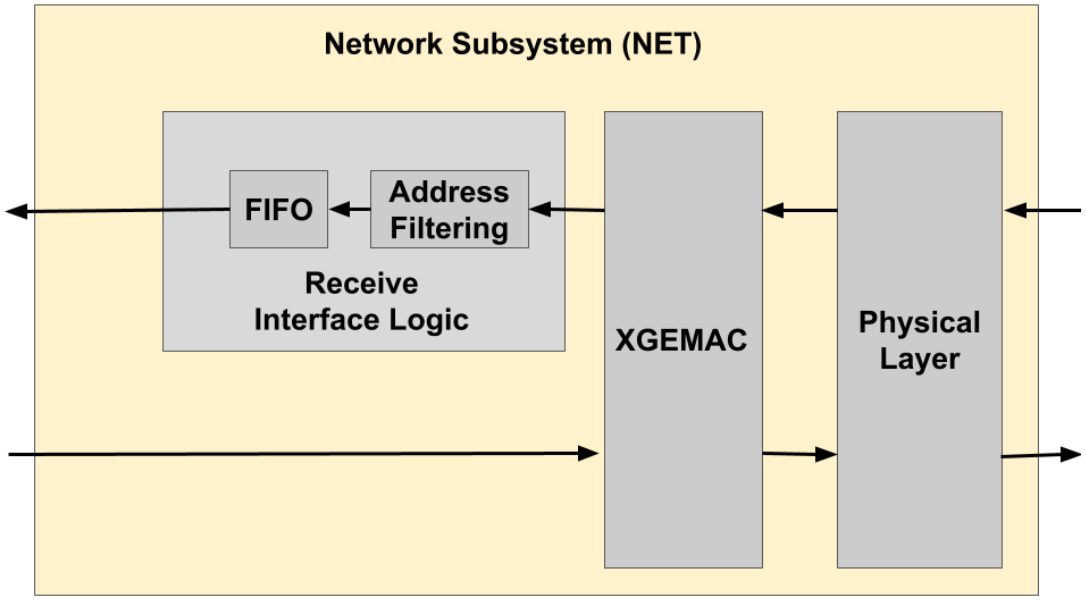


Figure 2.14: Net schematic.

The DMA IP (Figure 2.11 and Figure 2.12) is provided by Northwest Logic. This DMA controller is configured to support simultaneous operation of four user applications utilizing eight channels in total. This involves four system-to-card (S2C) or transmit channels and four card-to-system (C2S) or receive channels. In this work, the front-end of the DMA connects with the AXI4-Stream interface on the PCIe Endpoint. The backend of the DMA provides an AXI4-Stream interface as well, which connects to the user application side.

CONF: The Configuration Registers (Figure 2.11) are used to read and write control/status information to/from the FPGA components. This information ranges from configuring the network module to reading performance, power, and temperature information. There

is also a free address range available that is used to store configuration information to the specific IP-cores used in this work.

VFIFO: The TRD uses DDR3 space to implement a Virtual FIFO (VFIFO Figure 2.11). It realizes the VFIFO by means of the the following IP cores: (a) AXI Stream Interconnects (AXIS-IC Figure 2.13) that are used for width conversion and clock domain crossing as well as providing interconnect between the applications and the AXI VFIFO Controller; (b) AXI VFIFO Controller used for connecting the AXI Interconnect AXI-ST interface to AXI-MM interface on MIG, while also handling the addressing needs for the DDR3 FIFO; (c) Memory Interface Generator that provides the DDR3 memory controller for interfacing to external memory. This VFIFO is used to avoid back-pressure to the PCIe/DMA modules.

Network Subsystem: This subsystem is composed of four NET modules (see Figure 2.11 and Figure 2.14) containing each XGEMAC module and logic required to enable the physical layer of the Multi-FPGA communication network. Figure 2.14 shows the schematic for one channel. Each XGEMAC module receives data in the MAC frame format and sends it to the physical layer. Each NET module is connected to an SFP port capable of handling 10Gb/s per channel, resulting in a total of 40Gb/s bandwidth for the board. To receive data, the XGEMAC module receives and delivers to the *receive interface logic* (Figure 2.14) that filters the data according to the destination MAC address.

The next chapter shows how all of these components were used to implement the ideas of the work of this thesis.

Chapter 3

The OpenMP Multi-FPGA Infrastructure

This chapter describes the central idea of this thesis, which is to enable OpenMP Task Parallelism on Multi-FPGA Clusters. This work has four main goals: (a) to design a new Clang/LLVM plugin that understands FPGA boards as OpenMP devices, and uses OpenMP *declare variant* directive to specify hardware IP-core; (b) to create a mechanism based on the OpenMP task dependence and computation offloading model that enables transparent communication of IP-core in a Multi-FPGA architecture; (c) to design a hardware architecture, based on the Target Reference Design of the VC709 board, capable of executing OpenMP tasks using pre-existing IP-cores; (d) to enable a programming model based on OpenMP task parallelism, which makes it simple to move data among FPGAs, CPUs or other acceleration devices (e.g. GPUs), and that allows the programmer to use a single programming model to run its application on a truly heterogeneous architecture. The following two sections detail how these goals have been achieved.

3.1 Extending OpenMP

To explain how the proposed system works, please consider the code fragment shown in Listing 3.1. The code shown is a program that creates tasks to calculate iterations of stencil operations. Part of the solutions proposed in this thesis uses this application as a case of study. However, this application also shows some insights for more generic solutions that are discussed later in the text.

The *declare variant* directive (line 1), which is part of the OpenMP standard, declares a specialized hardware (FPGA IP-core) variation (*hw_ laplace2d*, in line 3) of a C function (*do_ laplace2d*, in line 2), and specifies the context in which that variation should be called. For example, line 2 of Listing 3.1 states that the variant *hw_ laplace2d* should be selected when the proper *vc709* device flag is provided to the compiler at compile time. This compiler flag is matched by the *match (device=arch(vc709))* clause and when the call to function *do_ laplace2d(&V, x, y)* (line 15) is to be executed, a call to the IP-core *hw_ laplace2d(int*, int, int)* is performed instead.

As shown in line 10 of Listing 3.1, the *main* function creates a pipeline of N tasks.

```
1  #pragma omp declare variant
2    (void do_laplace2d(int*,int,int))  match (device=arch(vc709))
3  extern void hw_laplace2d(int*,int,int);
4
5  int main(){
6    float  V[h*w];
7    bool deps[N+1];
8    #pragma omp parallel
9    #pragma omp single
10   for (int i = 0; i < N; i++){
11     #pragma omp target map(tofrom:V[:(h*w)]) \
12     depend(in:deps[i]) depend(out:deps[i+1]) \
13     nowait
14     {
15       do_laplace2d(&V,h,w);
16     }
17   }
18 }
```

Listing 3.1: Offloading task computations to FPGA IPs-cores.

Each task receives a vector V containing *h*w* (height and width) grid elements that are used to calculate a Laplace 2D stencil. As the tasks are created within a target region and the *vc709* device flag was provided to the compiler, the *hw_laplace2d* variant is selected to run each task in line 15. The compiler then uses this name to specify and offload the hardware IP-core that will run the task. As a result, at each loop iteration, a hardware IP task is mapped inside the FPGA. Details of the implementation of the Laplace 2D IP and other stencil IP-cores used in this work are provided in Section 4.1.

The *map* clause (line 11) specifies that the data is mapped back and forth to the host at each iteration. However, the implemented mapping algorithm concludes that vector V is sent to the IP-core from the host memory and its output forwarded to the next IP-core in the following iteration. The interconnections between these IP-core are defined according to the *depend* clauses (line 12). In the particular example of Listing 3.1, given that the dependencies between the tasks follow the loop iteration order, a simple pipeline task graph is generated. A careful comparison of Listing 3.1 with Listing 2.7 reveals that in terms of syntax, the implemented solution does not require the user to change anything concerning the OpenMP standard, besides specifying the *vc709* flag to the compiler. This gives the programmer a powerful verification flow. He/she can write the software version of *do_laplace2d* for algorithm verification purpose, and then switch to the hardware (FPGA) version *hw_laplace2d* by just using the *vc709* compiler flag.

To achieve this level of alignment with the OpenMP standard, three extensions were required to the OpenMP runtime implementation: (a) insertion of offloading information during the construction of the task graph; (b) modification of the task graph management mechanism; and (c) the design of the VC709 plugin in the libomptarget library. Figure 3.1 shows the stack of software that an application passes through. At the top is the program, just below the OpenMP runtime where the task graph is built and in the last two layers the libraries responsible for mapping and offloading. They are detailed below.
Task Graph Creation: Firstly, it was necessary to change the LLVM front-end, called

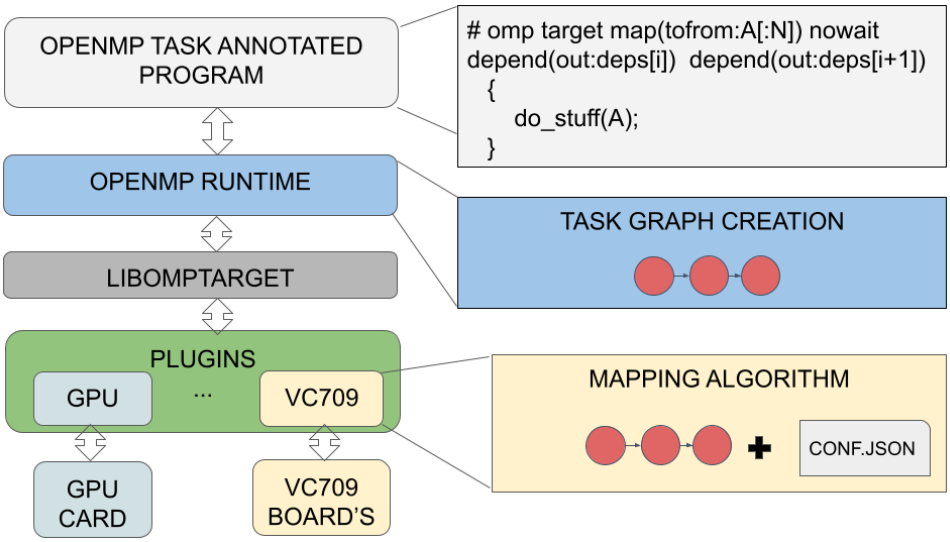


Figure 3.1: OpenMP software stack with VC709 plugin.

clang, so that when creating the task, information related to the offloading was passed on. This information are the variables to be mapped, the direction of the mapping (to, from or tofrom), and the size of the variables. Specifically for the work of this thesis, where the programmer needs to specify the IP-core to be used, it was also necessary to pass the name of the IP-core on to the runtime for creating the task graph (see Figure 3.1). As mentioned earlier, this is done using the *declare variant* directive. At compile time, the clang front-end captures this information and transmits it to the OpenMP runtime. Implementations details on this mechanism are given in Appendix A.1.

Managing the Task Graph. Another modification was related to how the OpenMP Runtime handles the task graph. In the current OpenMP implementation, the graph is built and consumed at runtime. Whenever a task has its dependencies satisfied, it is available for a worker thread to execute. After the worker thread finishes, the task output data is sent back to the host memory, as shown in Section 2.1.1. This approach satisfies the needs of a single accelerator, but causes unnecessary data movements for a Multi-FPGA architecture as the output data of one (FPGA) task IP-core may be needed as input to another task IP-core. To deal with this problem, the OpenMP runtime was changed so that tasks are not immediately dispatched for execution as they are detected by the control thread. In the case of FPGA devices, the runtime waits for the construction of the task graph at the synchronization point at the end of the scope of the OpenMP *single* clause (line 17 of Listing 3.1). That is, tasks are created but are not immediately consumed by worker threads. At the end of the single region, the entire task graph for the FPGA is available to be processed and executed, which is done by the VC709 plugin. More details on how this is implemented are given in Appendix A.2.

Building the VC709 Plugin. In the OpenMP implementation of the Clang/LLVM compiler [73], kernel/data offloading are performed by a library called *libomptarget* [26] (see Figure 3.1). This library provides an agnostic offloading mechanism that allows the insertion of a new device to the list of devices that the OpenMP runtime supports and is responsible for managing kernel and data offloading to acceleration devices. Therefore, to

```
{"#ofnodes" : 2,
  "nodes"   :
  [{"addr"  : "10.10.10.10" ,
    "bit"   : "file0.bit"    ,
    "#ips" : 3,
    "ips" [{"mac":"mac-addr0", "name":"ip0"},
           {"mac":"mac-addr1", "name":"ip1"},
           {"mac":"mac-addr2", "name":"ip2"}
          ]
   },
   {"addr"  : "10.10.10.11" ,
    "bit"   : "file1.bit"    ,
    "#ips" : 3,
    "ips" [{"mac":"mac-addr0", "name":"ip0"},
           {"mac":"mac-addr1", "name":"ip1"},
           {"mac":"mac-addr2", "name":"ip2"}
          ]
   }
  ]
 }
```

Listing 3.2: JSON file example.

allow the compiler to offload to the VC709 board, it was necessary to create a plugin in this library. Figure 3.1 illustrates where the plugin is located in the software stack. More details on how the plugin is implemented are given in Appendix A.3.

As shown in Figure 3.1, the plugin receives the task graph generated by the runtime and maps these tasks to the available IP-cores in the cluster. The cluster configuration is passed through a *conf.json* file (see Listing 3.2), which contains: (a) the number of FPGAs nodes, (b) the address of the node, (c) the location of the bitstream files, (d) the number of IP-cores available in each FPGA, and (e) the mac address and name of each IP-core. In the experiments that were conducted, the FPGAs are connected in a ring topology as shown in Figure 3.2. Of the four available optical interfaces, two are being used to connect each FPGA with its two respective neighbors. This organization is sufficient to execute the stencil applications. However, in a more generic solution, FPGAs can connect to an optical switch, which would allow complete communication between all of them.

A round-robin algorithm is used to map the tasks to the IP-cores. Each task is mapped in a circular order to the free IP-core that is closest to the host computer. The pseudo-code in Algorithm 1 shows how the mapping of the tasks to the IP-cores is done.

The algorithm starts with the *cluster::map* function (line 1), which belongs to the *cluster* class that contains all the information that was passed in the JSON configuration file (an example is shown in Listing 3.2), that is, this class holds information on how many FPGAs are there in the cluster, and how many and which IP-cores are there in each FPGA. The map function calls the *PARSER* function with the task graph root (line 2). The *PARSER* function uses the *GETIP* function (line 4) to search for an IP-core that is capable of executing the task that was passed as a parameter. The *GETIP* function (line 11) runs through all the nodes of the cluster (*for* loop in line 13), where each node

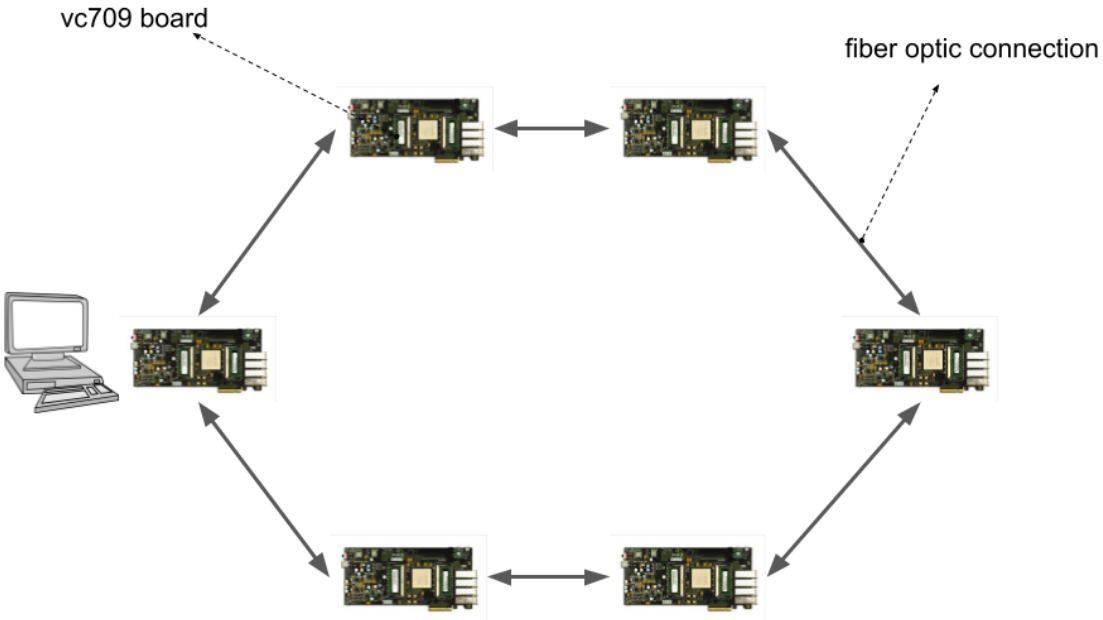


Figure 3.2: FPGAs organized in a ring topology.

is an FPGA. The *CLUSTER* variable (line 13) was filled with information from the json file. For each node, it runs through all the IP-cores available on that node (*for* loop in line 14).

As soon as an IP-core that can execute the task in question is found (*if* condition in line 15), that task is mapped to that IP-core. The *MATCH* function of the IP-core (line 15) checks if it is able to perform the task passed as an argument and if it is the least used IP-core among those that can perform this task. If this condition is true, the function returns true, otherwise, returns false. The *MAP* function in line 16 update the data dependency relationship between the IP-cores, information that will be used to configure the communication of the IP-cores within the FPGA.

Note that IP-cores can be reused, thus the algorithm also keeps track of the number of times that the IP-core was used so that the round-robin algorithm is repeated (*while* statement in line 12). Back to the *PARSER* function, on line 5, the variable *PT* receives all the parents of *task*. The *foreach* loop, in line 6, goes through all these tasks calling the *SETTARGET* function to set the destination IP-core that was just found. Right after that, the variable *CT* receives all the children of *task*. The *foreach* loop of line 9, goes through all these children calling again the function *PARSE*, entering into recursion until the entire task graph is traversed. This implementation of the mapping algorithm worked well for the tested benchmark. However, it is expected that an algorithm that treats different cases should be used for more complex task graphs. This could create opportunities for possible optimizations.

With the information of which IP-cores were used and the dependencies among them, the VC709 plugin is then able to configure the internal structure of the FPGAs. This structure is presented in the following section.

Algorithm 1 The Round-Robin Mapping Pseudo Algorithm.

```
1: function CLUSTER::MAP(graph)
2:     PARSER(graph.root)

3: function PARSER(task)
4:     GETIP(task)
5:     ▷ Let PT = task.parents
6:     for each P ∈ PT do
7:         P.SETTARGET(task.ip)
8:     ▷ Let CT = task.children
9:     for each C ∈ CT do
10:        PARSER(C)

11: function GETIP(tsk)
12:    while true do
13:        for each ND ∈ CLUSTER do
14:            for each IP ∈ ND do
15:                if IP.MATCH(tsk) then
16:                    MAP(IP,tsk)
17:                    return
```

3.2 Hardware Infrastructure

Besides the extensions to OpenMP, an entire hardware infrastructure was designed to support OpenMP programming of Multi-FPGA architectures. This infrastructure leverages on the Target Reference Design (TRD) presented in Section 2.2, but could also be ported to other modern FPGAs. To facilitate its understanding, the design is described below using two perspectives: Single-FPGA execution and Multi-FPGA execution.

Single-FPGA Execution. As discussed above, when the acceleration device is an FPGA, OpenMP uses the FPGA IP-cores to run the OpenMP tasks, which are specified by the name of a predefined variant function. These IP-cores are separately designed by a standard FPGA toolchain (e.g. Vivado), using the flow shown in Section 2.2.2. To connect the IP-cores into the infrastructure, the designer just needs to ensure that they use the AXI4-Stream interface [96].

The AXI4-Stream is one of the many AMBA protocols designed to transport data streams of arbitrary width in hardware. Generally, 32-bit bus width is used, which means that 4 bytes get transferred during one cycle. For example, consider an FPGA with a 250MHz of programmable logic frequency. This yields a throughput of hundreds of megabytes per second depending on the memory management unit capabilities and configuration. Figure 3.3 shows the schematic of the handshake between two modules, a master and a slave, that implement the AXI4-Stream protocol. First, valid data is sent using the TDATA port, which is put together with the TLAST and TUSER control signals on the bus. Then, the TVALID signal is triggered indicating that valid data is ready to be sent. This signal is kept high until a response is sent from the slave module via the TREADY signal. Once this response is received, the data is transmitted in sync with the

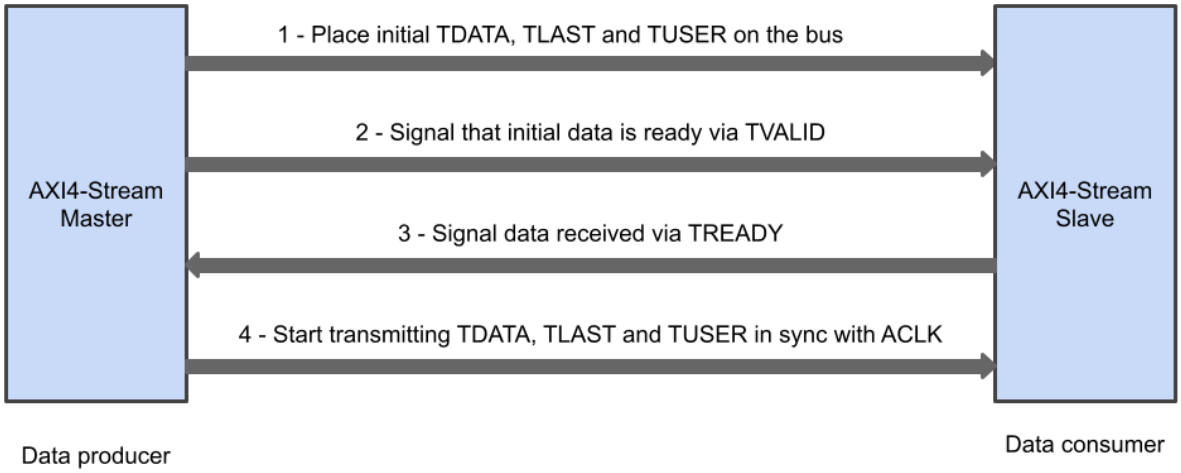


Figure 3.3: AXI4-Stream handshake.

ACLK clock.

TDATA width of bits is transferred per clock cycle. TLAST signals the last byte of the stream. It also has additional optional features: sending positional data with TKEEP and TSTRB ports which make it possible to multiplex both data position and data itself on TDATA lines; routing streams by TID and TDEST. The latter is used to route data between IP-cores within the FPGA. The infrastructure can be changed to accept other interfaces, although for the purpose of this work AXI-Stream suffices.

According to the proposed programming model, FPGA IP-cores can execute tasks that have dependencies among each other. In order to realize such dependencies an *AXI4-Stream Switch* module (A-SWT) was implemented. The A-SWT module is a hierarchy of switches that allows communication among the IP-cores and other modules within the FPGA. This enables the IP-cores to communicate directly to each other, based on the OpenMP dependencies programmed among them, thus avoiding unnecessary communication through the host memory. Each switch in the hierarchy is a AXI4-Stream Interconnect module [3] that enables the connection of heterogeneous master/slave AXI4-Stream protocol compliant endpoint IPs. The AXI4-Stream Interconnect routes connections from one or more AXI4-Stream master channels to one or more AXI4-Stream slave channels.

Figure 3.4 shows the schematic of an A-SWT module that allows the communication of 4 IP-cores, 4 DMA interfaces and 4 optical fiber interfaces. In the schematic of the figure, each IP-core only communicates with a neighboring IP-core, 2 DMA interfaces and 1 optical fiber interface. This topology was chosen because it meets the needs of the application used and best applied to the amount of resources available in the FPGA. However, this level of communication can be exploited according to the need and availability of resources, because the more connections the module has, the more resources will be used. It is also important to notice that the level of connectivity will also inter-

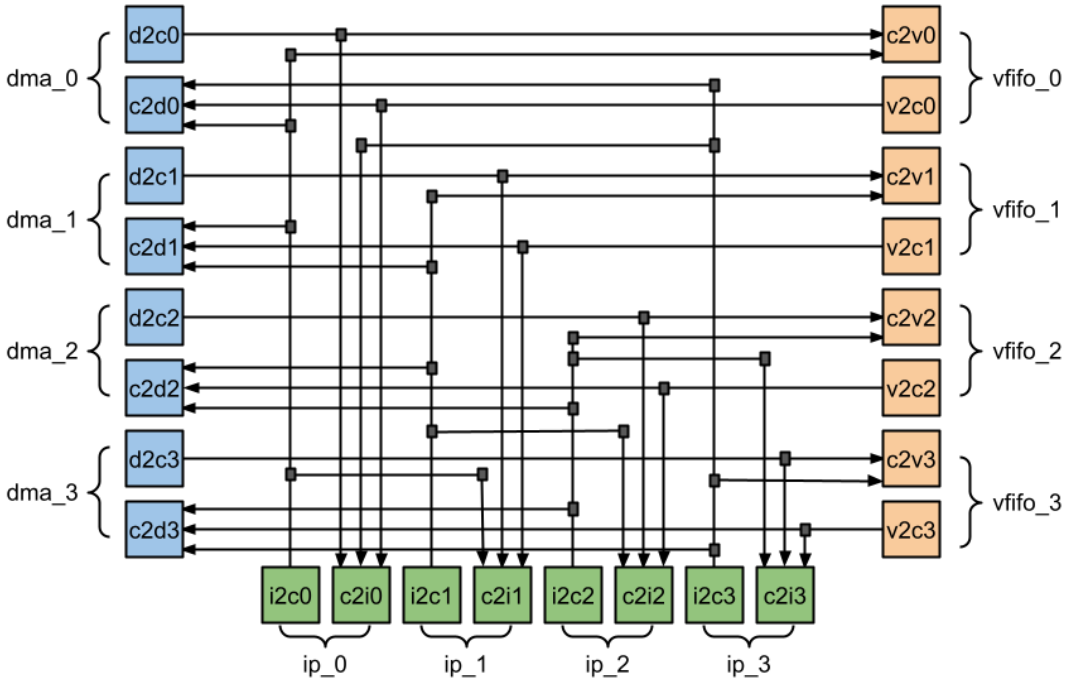


Figure 3.4: Integrated Block for AXI-Stream Interconnect RTL.

fere with the synthesis tool's place and routing algorithm, as shown in Section 2.2.2. In a modern FPGA, with more resources, the ideal is to have a switch fully connected allowing complete communication between the IP-cores.

The VC709 plugin uses the *CONF* register (Figure 3.5) bank to program the source and destination ports of each IP-core according to their specified task dependencies. That is, the registers are connected directly to the TDEST ports of the switches so that the plugin has full control of the data flow between the IP-cores and the other components of the FPGA.

Take Listing 3.1 again as example, but now creating 4 tasks (t0, t1, t2 and t3) that form a pipeline. If the infrastructure is composed of only one FPGA with four IP-cores capable of performing these tasks, the VC709 plugin, in possession of the graph and the cluster configuration, would configure the A-SWT module so that the tasks are mapped as shown in Figure 3.5. It is possible to notice that task 0 executes on IP 0, task 1 executes on IP 1, and so on, and data flows according to the dependencies specified in the task graph.

Multi-FPGA Cluster Execution. A Multi-FPGA architecture is composed of one or more cluster nodes containing at least one FPGA board each. To enable such architecture, routing capability needs to be added to each FPGA so that IP-cores from two different boards or nodes communicate through the optical links.

Therefore, to use the optical fibers, a module that can assemble and disassemble MAC frames is required. A MAC Frame Handler (MFH) module was designed and inserted into the hardware infrastructure, as shown in Figure 3.6. This module is required because the *Network Subsystem* that routes packages through the optical fibers receives data in the form of MAC Frames, which contain four fields: (a) *destination*, (b) *source*, (c) *type/length* and (d) *payload*. Figure 3.7 illustrates a MAC Frame.

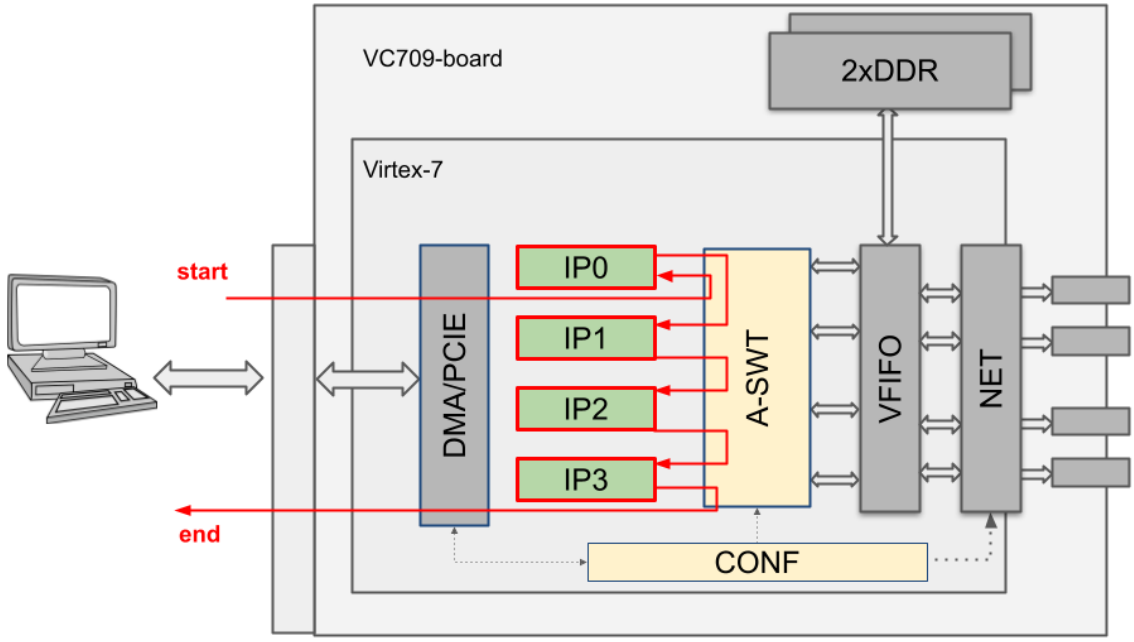


Figure 3.5: Single node execution.

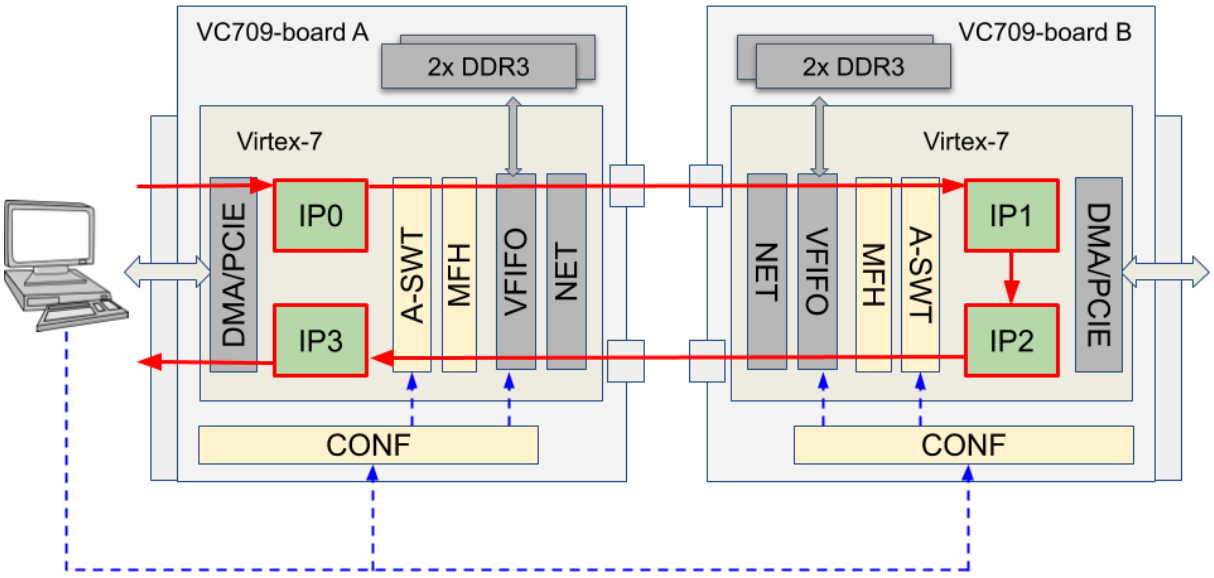


Figure 3.6: An optical-link interconnected Multi-FPGA architecture running an OpenMP pipelined application.

The MFH module is responsible for inserting and removing the source and destination MAC addresses and *type/lengh* fields whenever the IP-core needs to send/receive data through the *Network Subsystem*. MAC addresses are extracted from the dependencies in the task graph while the *type/lengh* fields are extracted from the *map* clause. The VC709 plugin uses this information to set up the *CONF* registers, which in turn configure the MFH module.

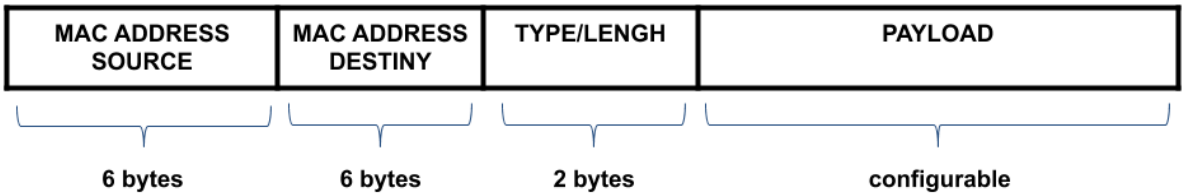


Figure 3.7: MAC Frame format.

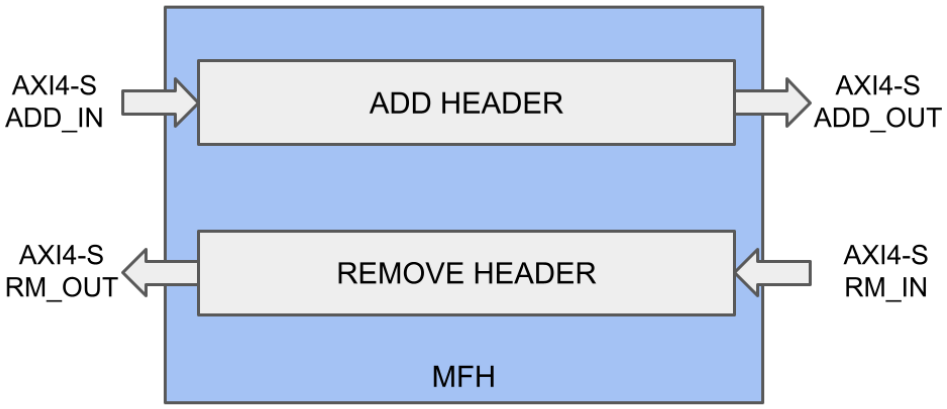
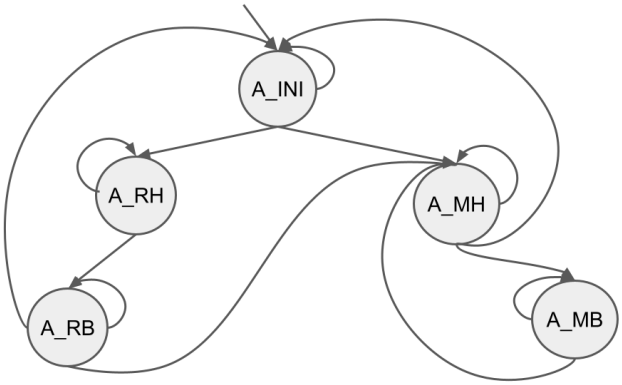


Figure 3.8: MAC Frame Handler schematic.

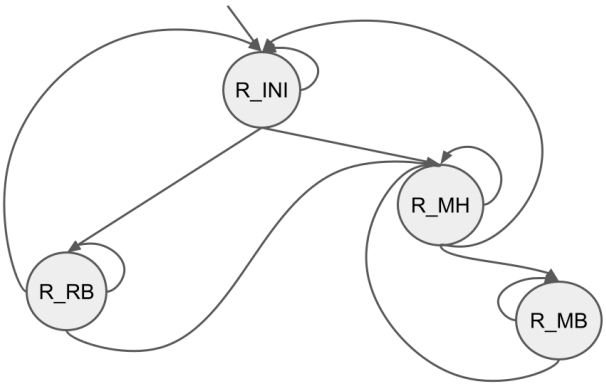
Figure 3.8 shows a simplified schematic of the MFH module. It consists of two pairs of AXI4-Stream ports, input and output. One pair is used to add the header (ADD_IN and ADD_OUT), so that the data goes out through the fiber, and the other is used to remove the header (RM_IN and RM_OUT), after the data arrives from the fiber. Figure 3.9a shows the simplified state machine for inserting a header. The A_INI state waits for the configuration that consists of the size of the buffer that will be sent. After receiving this size, the module is able to manipulate the data. For every 1024 bytes of data, the module inserts a header, with the size and destination MAC address, and a TLAST signal at the end of the frame. The header is inserted in the A_MH state, while the A_MB state bypasses the frame. For frames smaller than 1024, the A_RH state inserts the header and the A_RB state bypasses the payload.

A similar process occurs in the opposite direction. Figure 3.9b shows the state machine that performs the operation of removing the header and all the extra TLAST signals that were added by the dd Header State Machine. The R_MH state removes the header from the frames, while the R_MB state bypasses the payload removing the TLAST signals at the end of each payload, leaving only the original TLAST received from the software. The R_RB state removes the header from frames smaller than 1024 bytes.

With all of these components in place, the proposed VC709 plugin can distribute tasks to IP-cores across a cluster of FPGAs and map the dependency graph so that FPGA IP-cores communicate directly.



(a) Add Header State Machine.



(b) Rm Header State Machine.

Figure 3.9: MFH State Machines.

Consider again Listing 3.1, when 4 tasks (t0, t1, t2 and t3) are created to build a pipeline. This time the infrastructure is composed of two FPGAs with two IP-cores each capable of performing these tasks. The VC709 plugin would configure the A-SWT module and the MFH module so that the tasks are mapped as shown in Figure 3.6. It is possible to notice that task 0 executes on IP 0 on Board A, while task 1 executes on IP1 that seats on Board B; task 2 executes on IP2 still on Board B, and finally task 3 executes on IP3, now on board A. The data flows according to the dependencies specified in the task graph.

The next chapter shows, in details, the application of the proposed Multi-FPGA architecture and programming model when using stencil applications. It also describes the results of a set of experiments carried out in the proposed architecture.

Chapter 4

Experiments

4.1 An Stencil Multi-FPGA Pipeline

Stencil computation is a method where a matrix (i.e. grid) is updated iteratively according to a fixed computation pattern [88]. Figure 4.1 shows an example where an element of iteration *T+1* is calculated according to 4 elements of iteration *T*. Stencil computations are used in this work to show off the potential of the proposed OpenMP-based Multi-FPGA programming model. In this thesis, stencil IP-cores are used to process multiple portions and iterations of a grid in parallel on different FPGAs. There are basically two types of parallelism that can be exploited when implementing stencil computation in hardware: *cell-parallelism* and *iteration-parallelism* [93].

As detailed below, these two types of parallelism leverage on a pipeline architecture to improve performance and are thus good candidates to take advantage of the Multi-FPGA programming model described herein. Five different types of stencil IP-cores have been implemented for evaluation. The IP-cores were adapted from [93] and their computations are listed in Table 4.1 in the following order: (1) Laplace 2-D, (2) Diffusion 2-D, (3) Jacobi 9-pt. 2-D, (4) Laplace 3-D and (5) Diffusion 3-D. The formula in the *computations*

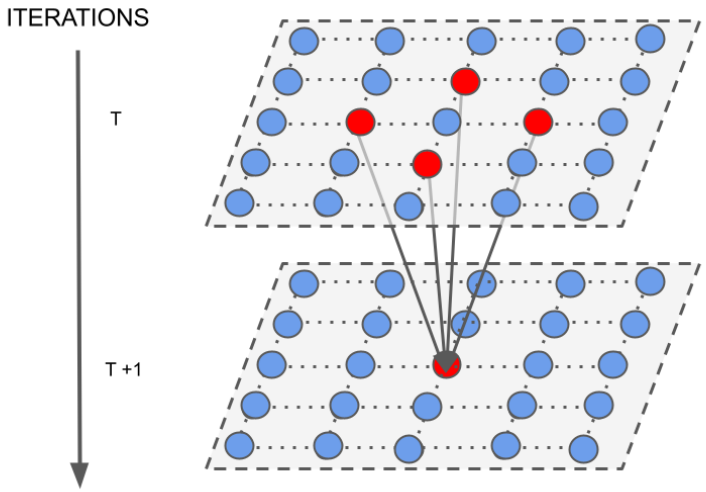
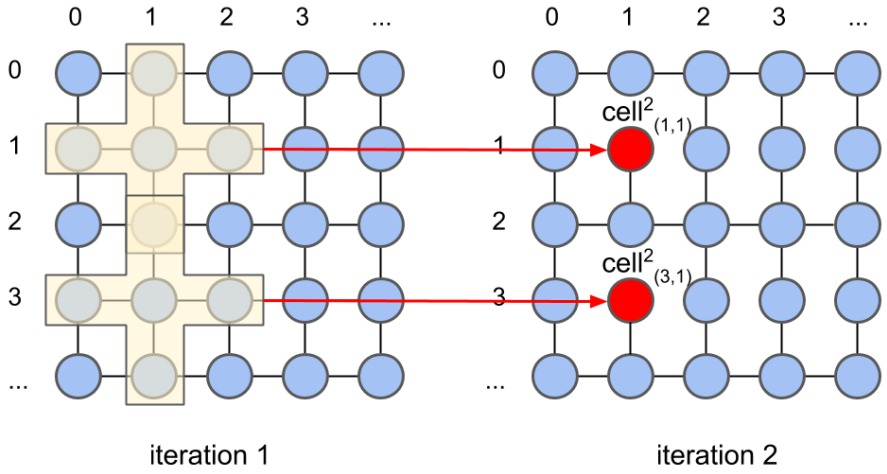
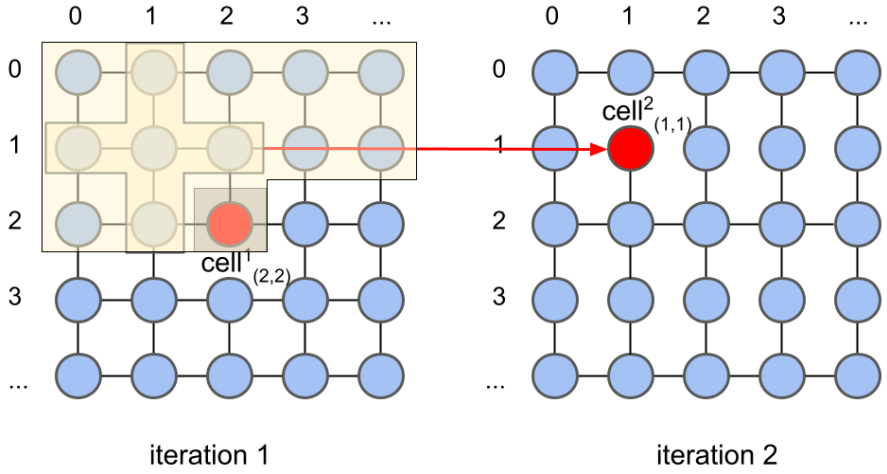


Figure 4.1: Stencil computation using 4-point 2-D stencil.



(a) Cell-Parallel computation.



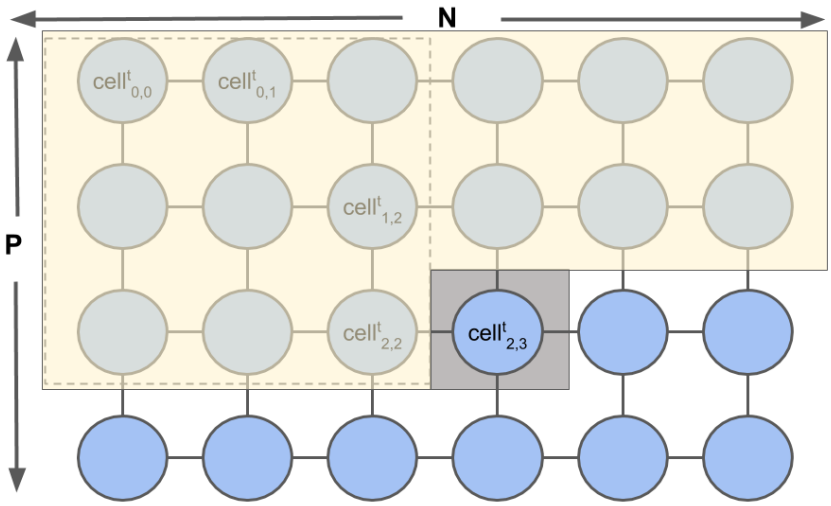
(b) Iteration-Parallel computation.

Figure 4.2: Types of stencil parallelism. Adapted from [93].

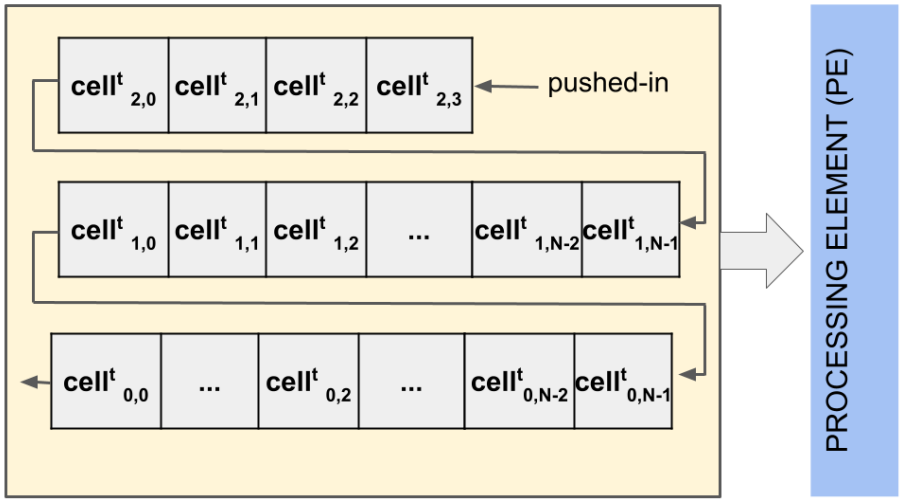
column is used to calculate an element $V_{i,j,k}^{t+1}$, where t represents the iteration and the indices i, j and k represent the axes of the grid. The C_* values are constants passed to the IP-cores.

Cell-Parallelism. Figure 4.2a shows an example of *cell-parallelism* on a stencil computation, where $cell^2_{(1,1)}$ at iteration 2 is computed using the data from its neighboring cells in the yellow area at iteration 1. This can be repeated for other cells at iteration 2, like $cell^2_{(3,1)}$ which is computed in parallel to $cell^2_{(1,1)}$.

Iteration-Parallelism. This occurs when elements of different iterations are calculated in parallel. Figure 4.2b shows two consecutive iterations (1 and 2) where this happens. As shown in Figure, $cell^2_{(1,1)}$ at iteration 2 is computed using the data from its neighboring cells in the yellow area at iteration 1 while at the same time, $cell^1_{(2,2)}$ from iteration 1 is also calculated. In this way, the elements of iteration 1 and 2 are calculated at the same time.



(a) Grid Stencil to be computed.



(b) Shift-register and PE implementation.

Figure 4.3: IP implementation. Adapted from [93].

4.1.1 IP-core Implementation

Figure 4.3 shows an overview of a typical stencil IP-core implementation using cell and iteration parallelism. Figure 4.3a shows the grid to be computed, and Figure 4.3b the components that implement the stencil, namely: (a) a *shift-register* that stores the grid data in processing order; and (b) the *processing element* (PE), which does the actual stencil computation. The cells in Figure 4.3a are computed by the architecture in Figure 4.3b from left-to-right and top-to-bottom, one after the other. At each clock cycle, data in the shift-registers are shifted to the left in Figure 4.3b, and a new cell value is pushed into the input of the first shift-register (i.e. $cell^t_{2,3}$). The computation starts after all neighboring data of a cell are available in the shift-register array. In the example of Figure 4.3, $cell^{t+1}_{1,1}$ is computed while input data is stored into $cell^t_{2,3}$. In the next clock cycle, the data at $cell^t_{0,0}$ at the output of the shift-register is discarded (shifted out), and the data of $cell^t_{2,4}$ is pushed into the input of the shift-register. Notice that the data at $cell^t_{0,0}$ is no longer required for any computation at this stage.

Kernel	Computations
Laplace 2D	$0.25(V^t_{i,j-1} + V^t_{i-1,j} + V^t_{i+1,j} + V^t_{i,j+1})$
Difussion 2D	$C_1.V^t_{i,j-1} + C_2.V^t_{i-1,j} + C_3.V^t_{i,j} + C_4.V^t_{i+1,j} + C_5.V^t_{i,j+1}$
Jacobi 9-pt	$C_1.V^t_{i-1,j-1} + C_2.V^t_{i,j-1} + C_3.V^t_{i+1,j-1} + C_4.V^t_{i-1,j} + C_5.V^t_{i,j} +$ $C_6.V^t_{i+1,j} + C_7.V^t_{i-1,j+1} + C_8.V^t_{i,j+1} + C_9.V^t_{i+1,j+1}$
Laplace 3D	$0.25(V^t_{i,j-1,k} + V^t_{i-1,j,k} + V^t_{i+1,j,k} + V^t_{i,j+1,k} + V^t_{i+1,j,k} + V^t_{i,j+1,k})$
Difussion 2D	$C_1.V^t_{i,j-1,k} + C_2.V^t_{i-1,j,k} + C_3.V^t_{i,j,k-1} +$ $C_4.V^t_{i,j,k} + C_5.V^t_{i+1,j,k} + C_6.V^t_{i,j+1,k}$

Table 4.1: Stencil kernels.

Each stencil IP-core has a *shift-register* and eight processing elements and is thus capable of processing up to eight elements at a time until the end of an iteration. Each IP-core works with a 256-bit AXI4-Stream interface, as each cell in the matrix is a 32-bit float.

The A-SWT switch in the architecture of Figure 1.1 can be configured so that the IP-cores can be reused, thus expanding the system's capacity to deal with larger grids and iteration counts. By doing so, the stencil pipeline can be scaled in both space and time. Such scaling is required to leverage the processing power of the multiple FPGAs, and to enable the computation of large-size problems that could not be done by a single FPGA due to the lack of resources. However, as discussed in Section 4.2, the size and number of IP-cores in an FPGA is constrained by the ability of the synthesis tool and designer to make efficient usage of the FPGA resources, and this sometimes can become a bottleneck.

To evaluate the presented system, four sets of experiments were performed using the stencil IP-cores described in Table 4.1. The first set (Section 4.2) aimed at evaluating the scalability of the system concerning the number of FPGAs. For the second set of experiments (Section 4.3) the scalability concerning the number of IP-cores (i.e. number of iterations) was evaluated. The goal of the third set of experiments (Section 4.4) was to evaluate FPGA resource utilization. Finally, in the fourth experiment, the Laplace 2D and Laplace 3D IP-cores were executed on a single FPGA without the VFIFO and Network Subsystem modules. This experiment was done to verify how the synthesis tool behaves in a situation with fewer support modules, and thus less placement and routing restrictions. The goal of this experiment was to evaluate how many IP-cores would be possible to fit into a modern/larger FPGA.

For all experiments, the board used was the Virtex-7 FPGA VC709 Connectivity Kit [14]. Compilation of the HDL codes was done using Vivado 2018.3 [12].

Infrastructure issues. The experiments did not aim for raw performance numbers but to demonstrate the viability and scalability of the proposed programming model. The infrastructure used in the experiments is not new. It has old Intel Xeon E5410 @2.33GHz CPUs, DDR2 667MHz memories, and archaic PCIe gen1 interfaces, which caused a considerable loss of performance since the FPGA boards use PCIe gen3. Moreover, as detailed in Section 4.4, the size of the original TRD kit made it very hard for Vivado to synthesize more IP-cores per FPGA, thus reducing the number of grid points inside the hardware,

Stencil Name	Grid Size	Iterations	# IP-cores
Laplace 2D	4096x512	240	4
Laplace 3D	512x64x64	240	2
Difussion 2D	4096x512	240	1
Difussion 3D	256x32x32	240	1
Jacobi 9-pt. 2-D	1024x128	240	1

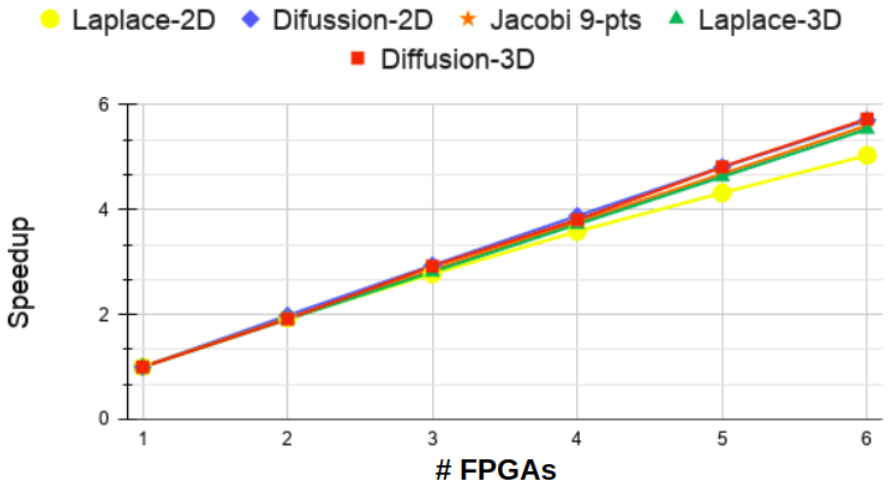
Table 4.2: The setup of the stencil IP-cores.

and the number of iterations. This harmed the final FPGA utilization and overall performance. However, even under these drawbacks the presented approach still achieved almost linear speedups. Therefore, we are confident that after using more modern machines and FPGAs (e.g. U250) the resulting performance will be very competitive to that shown in the hand-designed solution of [93], which in some cases surpasses the performance of GTX 980 Ti and P100 GPUs.

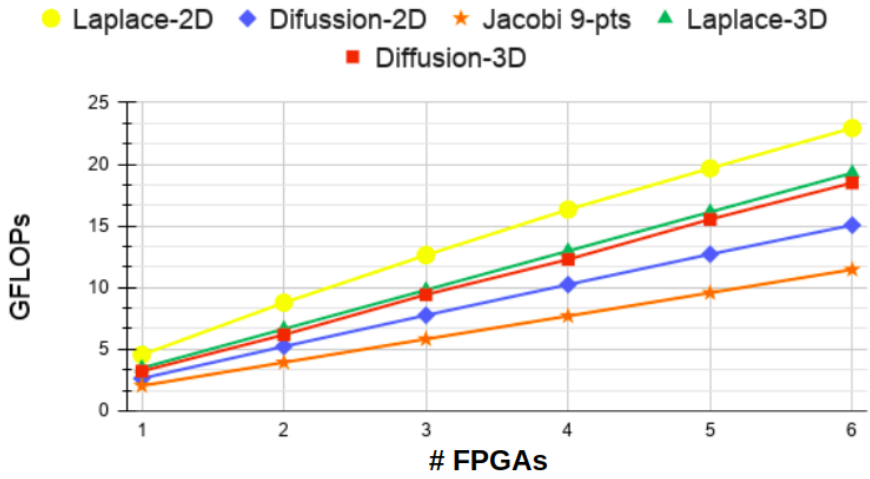
4.2 FPGA Scalability

The FPGA's scalability experiments were conducted with the settings shown in Table 4.2, and varying the number of FPGAs from 1 to 6. The *Grid Size* column shows the dimensions of the initial grid for each kernel. The more computation a kernel does, the more difficult it was for Vivado to synthesize the design under the time constraints. These constraints are related to performing placement and routing (as described in Section 2.2.2), and aim at ensuring that the three clock domains (250 MHz for user, 200 MHz for memory access and 156.25MHz for the optical fiber) are distributed among the sources of the clocks and the modules that need them. For this reason, the dimensions of the grid at each kernel were adjusted to avoid negative slacks. The *Iterations* column was set at 240 so that it was possible to execute with all 6 FPGAs. The *# IP-cores* column specifies the number of IP-cores at each FPGA. The number of IP-cores varies for the same reason as the dimensions of the initial grid. The larger the kernel computation, the smaller the number of IP-cores synthesized. On the other hand, as discussed in Section 4.4 there is still plenty of hardware to be used before the FPGA runs out of resources, which reinforces the long term potential of the proposed model.

The graph of Figure 4.4a shows the speedup, concerning the execution on a single FPGA, achieved by the various stencil kernels, as the number of FPGAs varies on the x-axis. The speedup grows almost linearly with the number of FPGAs for all five kernels. This result shows that it is possible to scale applications using Multi-FPGA architectures by using programming models like the one presented in this work to facilitate the design of such systems. The graph of Figure 4.4b shows, on the y-axis, the number of floating-point operations (GFLOPs) for each kernel as the number of FPGA varies on the x-axis. The Laplace-2D kernel (yellow line) executes more GFLOPs than the other kernels. Although the computation of this kernel is the simplest one, during synthesis it was possible to insert more IP-cores (four) per FPGA, which allowed more iteration parallelism as discussed in



(a) Speedup scaling with the number of FPGAs.



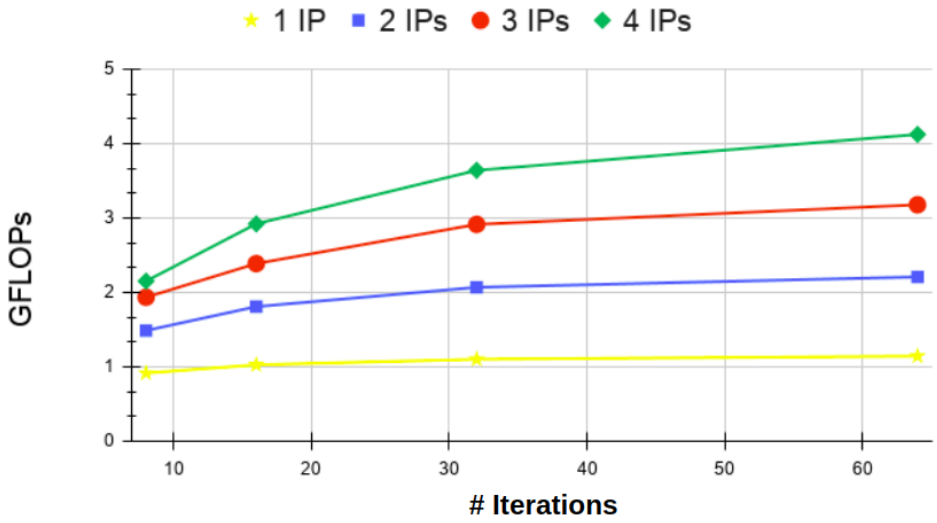
(b) GFLOPS scaling with the number of FPGAs.

Figure 4.4: FPGA Scalability Experiments.

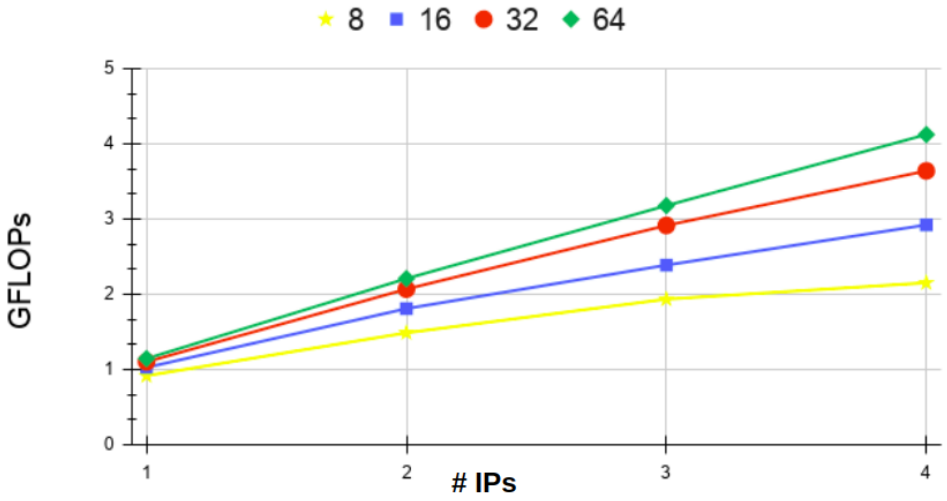
Section 4.1. Just below the Laplace-2D is the Laplace-3D (green line); with only 2 IP-cores per FPGA it still managed to sustain a linear performance growth. For the remaining kernels, as they all have only one IP-core per FPGA, the number of GFLOPs is related to the number of operations executed and the grid's dimensions. Notice that Diffusion-3D (red line) and Diffusion-2D (blue line) perform less computation than the Jacobi 9-pts (orange line). However, they achieve better GFLOP numbers due to their higher grid dimensions, thus enabling them to take advantage of the increased iteration parallelism.

4.3 Iteration and IP-core Scalability

A second experiment was performed to evaluate the IP-core scalability concerning the number of iterations. The Laplace-2D kernel was used as an example, although similar results have also been achieved for the other kernels. The graph in Figure 4.5a shows, on the y-axis, the number of GFLOPs produced by the system, as the number of iterations



(a) Laplace-2D scaling with the number of iterations.



(b) Laplace-2D scaling with the number of IP-cores.

Figure 4.5: Iteration Scalability Experiments.

Stencil	Slice LUTs		Block RAM		DSP	
	#	%	#	%	#	%
1	12138	7,5%	8	0,7%	16	0,4%
2	25024	15,4%	8	0,7%	80	2,2%
3	45733	28,3%	8	0,7%	144	4,0%
4	21790	13,5%	65	6,0%	17	0,5%
5	27615	17,1%	23	2,1%	97	2,7%

Table 4.3: IP-cores resource usage.

varies on the x-axis. The yellow, blue, red, and green lines represent executions with 1, 2, 3, and 4 IP-cores, respectively. As shown, the execution with a single IP (yellow line) remains practically constant.

On the other hand, the execution with 4 IP-cores shows an increase in performance

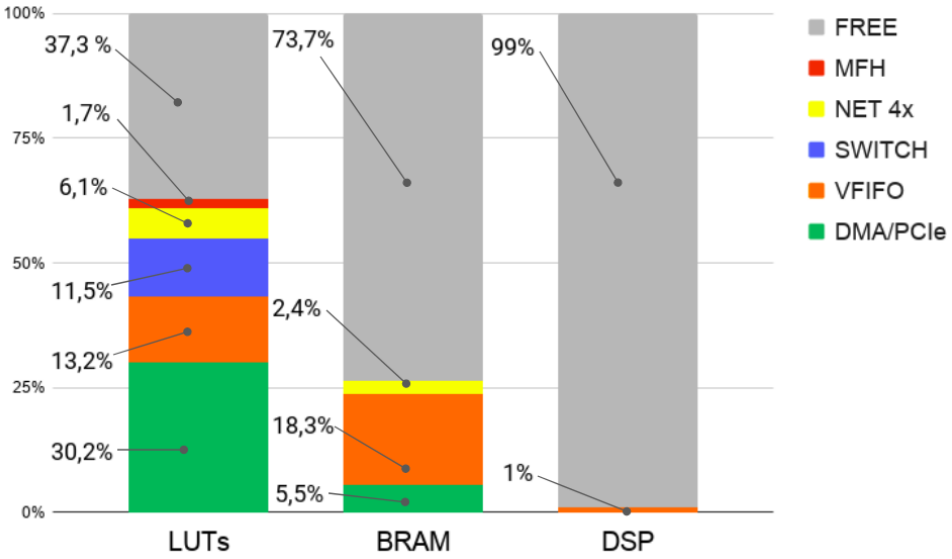


Figure 4.6: Resource usage distribution of the FPGA hardware.

until reaching a plateau. The executions with 2 and 3 IP-cores also show a gradual performance increase. This experiment reveals that by increasing the number of IP-cores, it is possible to improve the system's scalability in terms of iterations.

The graph of Figure 4.5b shows on the y-axis the number of GFLOPs for the Laplace-2D kernel as the number of IP-cores increase (x-axis). Each line in the graph is a different number of iterations. The graph reinforces the insight revealed in Figure 4.5a: as more IP-cores are added to the system, the more significantly the increase in the number of iterations improves performance. This can be confirmed by looking at the distances between the lines in Figure 4.5b, which grow larger as the number of IP-cores increase. This experiment also supports the case for Multi-FPGA architectures.

4.4 Resource Utilization

Regarding resource utilization, the graph in Figure 4.6 shows the percentage of occupancy of the FPGA main components of the proposed architecture (not considering the IP-cores). Remarkably, the DMA/PCIe component occupies 30.2% of the available LUTs. This large utilization comes from the fact that the DMA/PCIe was designed to support a board with four communication channels, although the proposed approach just requires one. Components MFH, SWITCH, VFIFO, and Network occupy, respectively, 1.7%, 11.5%, 13.2% , and 6.1% of the available LUTs. BRAMs are used by the DMA/PCIe (5.5%), VFIFO (18.3%), and NET (2.4%). The most significant usage of BRAMs comes from VFIFO, which uses it to multiplex and demultiplex the four channels of the virtual FIFO. DSP is the least used component (1%).

Table 4.3 shows the quantity and percentage of the FPGA components used by each IP from the *free* region (gray area) of Figure 4.6. The percentage of the available LUTs effectively used by the stencil IP-cores varies from 7.5% to 28.3%, depending on the complexity of the kernel. As for BRAM, the utilization ranges from 0.7% to 6.0%. This

Table 4.4: Synthesis without network infrastructure.

Stencil	Dimensions	# IP-cores	GFLOPs
Laplace 2D	4096x8000	16	22.6
Laplace 3D	256x128x1000	10	20.3

is directly linked to the size of the *shift-registers*, and is impacted by the size of the grid to be calculated. The number of DSP components used by the IP-cores varies from 0.4% to 4.0%, and is related to the number of multiplications performed by each kernel. The small utilization of the FPGA resources by the kernels has been previously discussed in the beginning of Section 4.2.

4.5 Single FPGA Synthesis

To verify how the synthesis tool would be impacted in a situation with less space restrictions on the FPGA, 2D and 3D Laplace stencils were executed on the FPGA infrastructure without the VFIFO, MFH and Network Subsystem modules. These three modules together occupied 21% of the LUTs available in the FPGA, a space that, in this experiment, was left free for the synthesis tool to better accommodate the stencil IP-cores. In this way, it was possible to synthesize more IP-cores, and compute larger grids. Table 4.4 shows that, for Laplace 2D, it was possible to synthesize 16 IP-cores, each computing a 4096x8000 dimension grid and reaching 22.6 GFLOPs. While for Laplace 3D, it was possible to synthesize 10 IP-cores, computing a 256x128x1000 dimension grid and reached GFLOps of 20.3. These results indicate that, using a more modern/larger FPGA (e.g. Alveo), the synthesis tool would be less limited by the space restriction, thus allowing the generation of more IP-cores that, combined with the scalability shown in the previous experiments, would allow for an improved performance.

Chapter 5

Related Works

Table 5.1 summarizes the main works found in the literature that present proposals for the use of OpenMP in FPGAs. Some of these works are also cited in [78] and in this section we discuss the characteristics that are most interesting and related to the work of this thesis. Each table row is a published work and each column lists a certain feature of the work. The papers are sorted by date published from oldest to newest. The table columns are:

- **HLS**: specifies whether or not the proposal uses High Level Synthesis [55], to synthesize high level programming language into hardware description language;
- **Compiler ToolKit**: specifies which compiler, framework or library the proposal uses.
- **OpenMP Task**: specifies whether or not the proposal works with OpenMP tasks.
- **Target System**: Specifies which platform the proposal was designed for.
- **Multi-FPGA**: specifies whether or not the proposal targets Multi-FPGA systems.

HLS is a research line widely found in the literature, our approach does not use HLS but pre-synthesized IP-cores, which in general are more efficient. The Compiler ToolKit also varies a lot between the works presented, we chose to use LLVM because it is widely adopted both in the industry and in the academy. The use of the OpenMP task is shown because we use this feature, along with the depend clause, to orchestrate the execution of the IP-cores. The System target shows the variety of platforms used in the literature. Finally, the Multi-FPGA column is the differential of our work which, as far as we know, is the first to integrate the parallelism of OpenMP tasks with Multi-FPGA systems. The paragraphs below discuss in more detail each work listed in Table 5.1.

Leow et al [75] proposes to use OpenMP as a hardware description language. Unlike other proposals, there is no differentiation between host code and device code, everything is synthesized into a single hardware. The authors use the C-Breeze compiler framework as a custom high-level synthesis pass to generate both Handel-C [6] and VHDL code [59]. Using the applications matrix multiplication and sieve of Eratosthenes, the FPGA versions achieve speedups of 25x and 7x over a symmetrical SMP UltraSPARC III with 8GiB [60]. For Mandelbrot, the FPGA version is slower than the SMP.

Table 5.1: Related works

	HLS	Compiler Toolkit	OpenMP Task	Target System	Multi FPGA
Leow 2006	NO	C-Breeze	NO	Celoxica RC100 Board with Spartan II	NO
Cabrera 2009	NO	Mercurium, GCC, SGI RASClib	YES	SGI RASC 2.2 Board with Virtex 4	NO
Cilardo 2013	N.A	Custom OpenMP Xilinx EDK	YES	Board Supported by Xilinx EDK	NO
Choi 2013	NO	LLVM GCC	NO	Altera Board with Stratix IV	NO
Filgueras 2014	YES	Mercurium, Nanos++	YES	Xilinx Board with Zynq-700	NO
Podobas 2014	NO	Custom C89 Compiler	YES	Altera ED5 Stratix V	NO
Sommer 2017	YES	Clang, LLVM, libomptarget, TPC	NO	Xilinx VC709 Board	NO
Ceissler* 2018	NO	Clang LLVM, libomptarget	NO	Amazon AWS Intel HARP 2	NO
Bosh 2018	YES	Mercurium, Nanos++	YES	Xilinx Board Zynq Ultrascale+	NO
Knaust 2019	YES	Clang LLVM	NO	Intel Board Arria 10 GX	NO
Huthmann 2020	YES	Clang/LLVM and libomptarget	NO	Intel FPGA PAC D5005 and Nymble	NO
Nepomuceno 2021	**NO**	**Clang/LLVM and libomptarget**	**YES**	**Multi-FPGA VC709 System**	**YES**

The work of Cabrera et al [29] is based on OpenMP 3.0 with some new extensions for task and target to ease the offloading of pre-synthesized hardware. There is no HLS, the hardware accelerator is built in a separate workflow. Their main contribution is that they provide support for SGI's RASC platform [10] and a multi-threaded runtime library layer with a bitstream cache that enables parallel computation on both the host and the FPGA. Offloading is implemented as a plugin for the Mercurium compiler [25]. The paper only shows runtimes of a matrix multiplication without any comparisons with CPU codes. The difference to our proposal is that they do not target multi-FPGAS systems and also do not deal with dependencies between tasks.

Cilardo et al in [40], uses OpenMP not only as a hardware description language, but as a language for describing a complete heterogeneous system (Xilinx Zynq [44]). They map the whole OpenMP program to the FPGA. The Xilinx Embedded Development Kit (EDK) [18] was used with the MicroBlaze [13] softcore for the sequential parts. The authors compare their sieve of Eratosthenes implementation with the results from [75] and they see twice the speedup. Furthermore, a runtime overhead inspection of the implemented OpenMP directives (private, firstprivate, dynamic, static, and critical) shows significantly less overhead than the SMP versions on an Intel i7 (6x, 1.2x, 3.1x, 10.5x, and 2.64x, respectively).

The system by Choi et al [38] aims to use the information provided by pragmas to generate better parallel hardware. The compiler synthesizes one kernel IP per thread in the source program. Nested parallelism is possible just in two levels. The extended LegUp [30] generates parallel hardware for *parallel* and *parallel for* and utilizes the other pragmas (atomic, etc.) to synchronize between the threads. With the best compiler configuration for the FPGA versions, benchmarks (Black-Scholes option pricing, simulated light propagation, Mandelbrot, line of sight, integer set division, hash algorithms, double-precision floating point sine function) show a geometric mean speedup of 7.6x and a geometric mean area-delay product of 63% compared to generated sequential hardware.

Filgueras et al in [54] proposes an extension to the OmpSs [4] framework to support the Xilinx Zynq FPGA [15] platform. Their prototype exclusively uses the FPGA's ASIC CPUs for the sequential portion of the source code, even so the authors claim any work distribution to be possible. The *task* pragma is extended so that it can be used to annotate functions and to specify dependencies between tasks (clauses in, out, or inout). They implemented their proposal using the Mercurium framework and the Nanos++ OpenMP runtime. On four numeric benchmarks (two matrix multiplications with different matrix sizes, complex covariance, and Cholesky decomposition) the FPGA version achieves speedups between 6x to 30x compared to a single ARM A9 core.

In the work proposed by Podobas et al, in [83] the compiler synthesizes an entire System on a Chip (SoC) based on task-annotated functions. The main program is rewritten to use such Soc, and the whole system is put into an FPGA using a softcore. The first implementation proposed by the authors built hardware for each task. However, they later fuse task kernel IP-cores for resource sharing. To evaluate the proposal they study three basic benchmarks (pi, Mandelbrot, and prime numbers). For the first two compute-bound benchmarks, the FPGA version outperforms both CPU-only versions (57-core Intel Xeon PHI and 48-core AMD Opteron 6172) by a factor of 2 to 3. However, for

the memory-bound third benchmark, the CPU versions are about 100 times faster.

Sommer et al [89] uses Vivado HLS to generate hardware from OpenMP target regions extracted from the source program. The authors use their Thread Pool Composer (TPC) [69] [19] API (now called TaPaSCo [70]) to implement the host-to-FPGA communication. The main point of the proposal is that it fully supports omp target (including its map clause). This project is also the first that integrated libomptarget. To evaluate the proposal they used six benchmarks from the Adept benchmark suite [95] and compared the runtime of -O3-optimized i7 CPU code (4 cores) to their FPGA-only version (with HLS pragmas). The CPU outperforms the FPGA version by 3x to 3.5x (without the HLS pragmas: 6x to 9x).

Ceissler et al [34] proposes HardCloud, an OpenMP platform that integrates FPGA acceleration to OpenMP. The authors propose three more clauses to the OpenMP 4.X standard: *use*, *check* and *module*. The *use* and *check* clauses are used to automatically validate the hardware accelerator, while the module clause specifies the bitstream used to configure the FPGA. Hardcloud does not use HLS, i.e. there is no outlining of code blocks. Instead, it makes pre-synthesized functional units for FPGAs easier to use in existing OpenMP code by automating data transfer and device control. To evaluate the proposal the authors used nine benchmark programs and achieved speedups on the HARP 2 platform between 1.1x and 135x.

OmpSsFPGA by Bosch et al [27] is based on the work proposed by Filgueiras et al in [54]. Memory on the accelerator is used for data sharing (streaming). Besides outlining code to the FPGA, this system also addresses GPU. Moreover, the tasks are dynamically scheduled onto the devices. The authors evaluate the work using three benchmarks: matrix multiplication, n-body, Cholesky decomposition. They compare the baseline runtime (measured on a CARM-A52 4 with 4 GB of shared memory) with their FPGA versions. For the Cholesky decomposition, the performance drops by about 2x. For n-body, the FPGA version is 15x faster. The matrix multiplication on the FPGA achieves 6x the GFLOP/s. The difference to our proposal is that they use HLS, while we use pre-synthesized hardware, and they also do not target multi-FPGAs systems.

Knaust's et al [67] uses Clang [5] to outline omp target regions at the level of the LLVM IR and feeds them into Intel's OpenCL HLS [17] to generate a hardware kernel for the FPGA. For the communication between host and FPGA, the proposal uses Intel's OpenCL API [7]. The authors passes the unrolled pragma to the underlying HLS. From the map clauses of the target pragma, only array sections are unsupported. To evaluate the proposal Two Sobel filters (unoptimized and optimized for FPGAs) run on a 4096x2160x8 bit matrix. The CPU-only version is compiled without -fopenmp. The pure optimized kernel for the FPGA is 4x as fast as one CPU core, but this can hardly amortize the cost of transfer and initialization.

The work by Huthmann et al in [62] presents an approach to OpenMP device offloading for FPGAs based on the LLVM compiler infrastructure and the Nymble HLS compiler. The automatic compilation flow uses LLVM IR for HLS-specific optimizations and transformation and for the interaction with the Nymble HLS compiler. Parallel OpenMP constructs are automatically mapped to hardware threads executing simultaneously in the generated FPGA accelerator and the accelerator is integrated into libomptarget to

support data-mapping. The difference to our proposal is that they also do not target multi-FPGAs systems and uses HLS.

Outside the OpenMP spectrum, there are other alternatives for programming heterogeneous systems, such as Intel's OneAPI [64]. This is a single, unified programming model that aims to simplify development across different hardware architectures: CPUs, GPUs, FPGAs, AI accelerators, and more. It is based on the Data Parallel C++ language, or DPC++ for short. This is a C-based open-source alternative to proprietary programming languages typically used to code for specific types of hardware, such as GPUs or FPGAs. However, such alternatives require the programmer to learn a new language and build solutions from scratch, which in many cases is not feasible.

To the best of our knowledge, and contrary to the previous works, which focused mostly on the synthesis and single FPGA architectures, this paper is the first to enable OpenMP task parallelism to integrate IP-cores into a Multi-FPGA architecture.

Chapter 6

Final Remarks and Conclusion

The authors in [62] argue that scaling OpenMP onto multiple FPGAs is an open question. They suggest that one could rely on OpenMP's accelerator directives, and treat each device as a discrete system with little to no access to other systems and create/include special hardware to (for example) support a shared-memory view across multiple FPGAs, or use tasks as containers that encapsulate produced/consumed data, that are exchanged among FPGAs. The work presented in this thesis goes straight to that point. The implemented architecture was able to provide a complete systematic solution, with software and hardware support, to run OpenMP tasks in a Multi-FPGA cluster connected by optical fibers.

The results presented in Chapter 4 shows a promising scalable behavior even with all the infrastructure drawbacks. Solving these infrastructure issues would allow for improvements in a few points. For example, replacing the interface that implements the AXI-Stream protocol with an implementation of the AXI Memory Mapped protocol would make the solution more complete. In the current state, the IP-core plays a passive role while starting communication with the host application. Thus, this change would permit the IP-core to start a transfer of data giving more flexibility and increasing the range of applications that can take advantage of the system.

Another possible improvement in this work would be the adoption of partial reconfiguration. This, together with AXI Memory Mapped protocol, would allow the use of more complex and efficient mapping-scheduling algorithms. Works as the one published in [41], already propose new models of representation and mapping-schedule algorithms capable of taking advantage of these partial reconfigurable FPGA environments. Such solutions can be incorporated into the system presented in this thesis in order to minimize the dynamic reconfiguration overhead while meeting the communication requirements among the tasks.

Finally, another direct consequence of this work is the integration of the proposed model with other types of accelerators, such as GPUs. The OpenMP programming model is generic enough to allow this straightforward integration. Such systems, that combine GPUs and FPGAs, are useful to solve problems that can be divided into components which can benefit from different architecture types. For example, the simulation of the primitive universe is one such application. The radiation transfer from spotlight and spatially distributed light strongly affect the birth of the first objects in space. ARGOT [68]

is an application that performs such simulation. These two phenomena have distinct computation characteristics and thus could benefit from the specific architectural features of GPUs and FPGAs. The method name ART is better suitable for FPGA computing, while the remaining of the simulation is better performed on a GPU.

Applications like this can greatly benefit from a truly heterogeneous environment and a programming model such as the one presented in this thesis.

6.1 Future works

As future work we plan to move the entire infrastructure to a cluster of Xilinx Alveo FPGAs [2]. This will allow us to improve all the bottlenecks found in this work regarding the resources available in the FPGAs, since the Alveos have more resources. For example, it is expected that we will be able to implement a fully connected A-SWT module which will allow more complex communication between IP-cores. In addition to being able to implement more IP-cores per FPGA. As a result, we also plan to investigate smarter mapping/scheduling algorithms that can implement optimizations to take full advantage of the system. Finally, we will connect all FPGAs to an optical switch that will also contribute to the flexibility and generality of the system.

Bibliography

[1] About us - openmp. https://www.openmp.org/about/about-us/. (Accessed on 04/19/2021).

[2] Alveo. https://www.xilinx.com/products/boards-and-kits/alveo.html. (Accessed on 05/28/2021).

[3] Axi4-stream interconnect v1.1 logicore ip product guide (pg035).

[4] Barcelona supercomputing center: The ompss programming model. https://pm.bsc.es/ompss. (Accessed on 10/03/2019).

[5] Clang c language family frontend for llvm. https://clang.llvm.org/. (Accessed on 05/04/2021).

[6] Handel-c synthesis methodology - mentor graphics. https://www.mentor.com/products/fpga/handel-c/. (Accessed on 10/04/2019).

[7] Intel sdk for opencl applications. https://software.intel.com/en-us/opencl-sdk. (Accessed on 11/12/2019).

[8] OpenMP 3.0 Specifications. https://www.openmp.org/wp-content/uploads/spec30.pdf. Accessed on Oct 13, 2019.

[9] OpenMP 4.5 Specifications. http://www.openmp.org/mp-documents/openmp-4.5.pdf. Accessed on Oct 13, 2019.

[10] Reconfigurable application-specific computing user's guide. https://irix7.com/techpubs/007-4718-004.pdf. (Accessed on 10/04/2019).

[11] Tasking terminology. https://www.openmp.org/spec-html/5.0/openmpsu5.html. (Accessed on 04/19/2021).

[12] Vivado design suite user guide: Release notes, installation, and licensing (ug973). (Accessed on 11/16/2020).

[13] Xilinx: Microblaze soft processor core. https://www.xilinx.com/products/design-tools/microblaze.html. (Accessed on 10/04/2019).

[14] Xilinx virtex-7 fpga vc709 connectivity kit. https://www.xilinx.com/products/boards-and-kits/dk-v7-vc709-g.html. (Accessed on 11/11/2020).

[15] Zynq-7000 soc data sheet: Overview (ds190). (Accessed on 10/03/2019).

[16] Amazon EC2 F1 Instances. `https://aws.amazon.com/ec2/instance-types/f1`, Nov 2019. [Online; accessed 25. Nov. 2019].

[17] Intel High Level Synthesis Compiler: Reference Manual, Sep 2019. [Online; accessed 21. Nov. 2019].

[18] Platform Studio and the Embedded Development Kit (EDK), Oct 2019. [Online; accessed 31. Oct. 2019].

[19] ThreadPollComposer, Oct 2019. [Online; accessed 15. Oct. 2019].

[20] Miron Abramovici and Jose T. De Sousa. *Journal of Automated Reasoning*, 24(1/2):5–36, 2000.

[21] Alex Aiken, Utpal Banerjee, Arun Kejariwal, and Alexandru Nicolau. *Instruction Level Parallelism*. 01 2016.

[22] Samuel Antao, Carlo Bertolli, Andrey Bokhanko, Alexandre Eichenberger, Hal Finkel, Sergey Ostanevich, Eric Stotzer, and Guansong Zhang. Openmp offload infrastructure in llvm. `https://lists.llvm.org/pipermail/llvm-dev/attachments/20150408/225ab427/attachment.pdf`. (Accessed on 04/29/2021).

[23] Brian Armstrong and Rudolf Eigenmann. Application of automatic parallelization to modern challenges of scientific computing industries. In *2008 37th International Conference on Parallel Processing*. IEEE, September 2008.

[24] M. M. Azeem, R. Chotin-Avot, U. Farooq, M. Ravoson, and H. Mehrez. Multiple fpgas based prototyping and debugging with complete design flow. In *2016 11th International Design Test Symposium (IDT)*, pages 171–176, Dec 2016.

[25] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta. Nanos mercurium: a research compiler for openmp. In *European Workshop on OpenMP (EWOMP'04). Pp*, pages 103–109, 2004.

[26] Carlo Bertolli, Samuel F. Antao, Gheorghe-Teodor Bercea, Arpith C. Jacob, Alexandre E. Eichenberger, Tong Chen, Zehra Sura, Hyojin Sung, Georgios Rokos, David Appelhans, and Kevin O'Brien. Integrating gpu support for openmp offloading directives into clang. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 5:1–5:11, New York, NY, USA, 2015. ACM.

[27] J. Bosch, X. Tan, A. Filgueras, M. Vidal, M. Mateu, D. Jiménez-González, C. Álvarez, X. Martorell, E. Ayguade, and J. Labarta. Application acceleration on fpgas with ompss@fpga. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 70–77, Dec 2018.

[28] Ravi Budruk, Don Anderson, and Ed Solari. *PCI Express System Architecture*. Pearson Education, 2003.

[29] D. Cabrera, X. Martorell, G. Gaydadjiev, E. Ayguade, and D. Jimenez-Gonzalez. Openmp extensions for fpga accelerators. In *2009 International Symposium on Systems, Architectures, Modeling, and Simulation*, pages 17–24, July 2009.

[30] A. Canis, J. Choi, Y.T. Chen, and H. Hsiao. Legup high-level synthesis. `http://legup.eecg.utoronto.ca/`. (Accessed on 10/06/2019).

[31] P. Cappello. Multicore processors as array processors: Research opportunities. In *IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP'06)*, pages 169–172, 2006.

[32] W. Carter, Ic Duong, R. Freman, H. Hsieh, Jason Y. Ja, J. Mahoney, N. Ngo, and S. L. Sac. A user programmable reconfigurable logic array. In *Proceedings of the IEEE Custom Integrated Circuits Conference (CICC)*, pages 233–235, 1986.

[33] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[34] C. Ceissler, R. Nepomuceno, M. Pereira, and G. Araujo. Automatic offloading of cluster accelerators. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 224–224, April 2018.

[35] Luis H. Ceze. *Shared-Memory Multiprocessors*, pages 1810–1812. Springer US, Boston, MA, 2011.

[36] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*, volume 10. MIT press, 2008.

[37] Subodha Charles and Prabhat Mishra. Reconfigurable network-on-chip security architecture. *ACM Trans. Des. Autom. Electron. Syst.*, 25(6), August 2020.

[38] J. Choi, S. Brown, and J. Anderson. From software threads to parallel hardware in high-level synthesis for fpgas. In *2013 International Conference on Field-Programmable Technology (FPT)*, pages 270–277, Dec 2013.

[39] Grigorios Chrysos, Euripides Sotiriades, Christos Rousopoulos, Kostas Pramataris, Ioannis Papaefstathiou, Apostolos Dollas, Agathoklis Papadopoulos, Ioannis Kirmitzoglou, Vasilis J. Promponas, Theocharis Theocharides, George Petihakis, and Jacques Lagnel. Reconfiguring the bioinformatics computational spectrum: Challenges and opportunities of FPGA-based bioinformatics acceleration platforms. *IEEE Design & Test*, 31(1):62–73, February 2014.

[40] A. Cilardo, L. Gallo, A. Mazzeo, and N. Mazzocca. Efficient and scalable openmp-based system-level design. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 988–991, March 2013.

[41] Juan Antonio Clemente, Ivan Beretta, Vincenzo Rana, David Atienza, and Donatella Sciuto. A mapping-scheduling algorithm for hardware acceleration on reconfigurable platforms. *ACM Trans. Reconfigurable Technol. Syst.*, 7(2), July 2014.

[42] Katherine Compton and Scott Hauck. Reconfigurable computing: A survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, June 2002.

[43] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. Understanding performance differences of fpgas and gpus: (abtract only). In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, pages 288–288, New York, NY, USA, 2018. ACM.

[44] Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, and Robert W. Stewart. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc.* Strathclyde Academic Media, UK, 2014.

[45] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[46] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, March 1966.

[47] Apostolos Dollas. Reconfigurable architectures for bioinformatics applications. In *2010 IEEE Computer Society Annual Symposium on VLSI*. IEEE, July 2010.

[48] Donohoe, Gregory W., Lyke, and James C. Reconfigurable computing for space. In Thawar T. Arif, editor, *Aerospace Technologies Advancements*, chapter 3. IntechOpen, Rijeka, 2010.

[49] T. El-Ghazawi. Reconfigurable supercomputing. In *The IEEE/ACS International Conference onPervasive Services, 2004. ICPS 2004. Proceedings.*, pages 163–, 2004.

[50] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell. The promise of high-performance reconfigurable computing. *Computer*, 41(2):69–76, Feb 2008.

[51] Tarek El-Ghazawi. Is high-performance, reconfigurable computing the next supercomputing paradigm? In *ACM/IEEE SC 2006 Conference (SC'06)*. IEEE, November 2006.

[52] G. Estrin, B. Bussell, R. Turn, and J. Bibb. Parallel processing in a restructurable computer system. *IEEE Transactions on Electronic Computers*, EC-12(6):747–755, 1963.

[53] U. Farooq, I. Baig, and B. A. Alzahrani. An efficient inter-fpga routing exploration environment for multi-fpga systems. *IEEE Access*, 6:56301–56310, 2018.

[54] Antonio Filgueras, Eduard Gil, Daniel Jimenez-Gonzalez, Carlos Alvarez, Xavier Martorell, Jan Langer, Juanjo Noguera, and Kees Vissers. Ompss@zynq all-programmable soc ecosystem. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 137–146, New York, NY, USA, 2014. ACM.

[55] Michael Fingeroff. *High-Level Synthesis Blue Book*. Xlibris Corporation, 2010.

[56] Lin Gan, Ming Yuan, Jinzhe Yang, Wenlai Zhao, Wayne Luk, and Guangwen Yang. High performance reconfigurable computing for numerical simulation and deep learning. *CCF Transactions on High Performance Computing*, 2(2):196–208, June 2020.

[57] Philip Garcia, Katherine Compton, Michael Schulte, Emily Blem, and Wenyin Fu. An overview of reconfigurable hardware in embedded systems. *EURASIP Journal on Embedded Systems*, 2006:1–19, 2006.

[58] Z. Guo, T. W. Huang, and Y. Lin. Gpu-accelerated static timing analysis. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2020.

[59] Ulrich Heinkel, Wolfram Glauert, and M. Wahl. *The VHDL Reference: A Practical Guide to Computer-Aided Integrated Circuit Design (Including VHDL-AMS) with Other*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[60] T. Horel and G. Lauterbach. Ultrasparc-iii: designing third-generation 64-bit performance. *IEEE Micro*, 19(3):73–85, May 1999.

[61] Mingkai Hsueh. *Reconfigurable Computing for Algorithms in Hyperspectral Image Processing*. PhD thesis, USA, 2007. AAI3283819.

[62] Jens Huthmann, Lukas Sommer, Artur Podobas, Andreas Koch, and Kentaro Sano. OpenMP device offloading to FPGAs using the nymble infrastructure. In *OpenMP: Portable Multi-Level Parallelism on Modern Systems*, pages 265–279. Springer International Publishing, 2020.

[63] Qaiser Ijaz, El-Bay Bourennane, Ali Kashif Bashir, and Hira Asghar. Revisiting the high-performance reconfigurable computing for future datacenters. *Future Internet*, 12(4):64, April 2020.

[64] Intel. Intel oneapi: A unified x-architecture programming model. `https://software.intel.com/content/www/us/en/develop/tools/oneapi.html#gs.zo72sh`. (Accessed on 05/02/2021).

[65] Weiwen Jiang, Edwin H.-M. Sha, Xinyi Zhang, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. Achieving super-linear speedup across multi-fpga for real-time dnn inference. *ACM Trans. Embed. Comput. Syst.*, 18(5s):67:1–67:23, October 2019.

[66] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. *SIGARCH Comput. Archit. News*, 17(2):272–282, April 1989.

[67] M. Knaust, F. Mayer, and T. Steinke. Openmp to fpga offloading prototype using opencl sdk. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 387–390, May 2019.

[68] Ryohei Kobayashi, Norihisa Fujita, Yoshiki Yamaguchi, Taisuke Boku, Kohji Yoshikawa, Makito Abe, and Masayuki Umemura. Multi-hybrid accelerated simulation by GPU and FPGA on radiative transfer simulation in astrophysics. *Journal of Information Processing*, 28(0):1073–1089, 2020.

[69] Jens Korinth, David de la Chevallerie, and Andreas Koch. An open-source tool flow for the composition of reconfigurable hardware thread pool architectures. In *Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '15, pages 195–198, Washington, DC, USA, 2015. IEEE Computer Society.

[70] Jens Korinth, Jaco Hofmann, Carsten Heinz, and Andreas Koch. The tapasco open-source toolflow for the automated composition of task-based parallel reconfigurable computing systems. In Christian Hochberger, Brent Nelson, Andreas Koch, Roger Woods, and Pedro Diniz, editors, *Applied Reconfigurable Computing*, pages 214–229, Cham, 2019. Springer International Publishing.

[71] D. M. Kunzman and L. V. Kale. Programming heterogeneous systems. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 2061–2064, May 2011.

[72] Marco Lanzagorta, Stephen Bique, and Robert Rosenberg. Introduction to reconfigurable supercomputing. *Synthesis Lectures on Computer Architecture*, 4(1):1–103, January 2009.

[73] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[74] S. Lee, J. Kim, and J. S. Vetter. Openacc to fpga: A framework for directive-based high-performance reconfigurable computing. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 544–554, May 2016.

[75] Y. Y. Leow, C. y. Ng, and W. f. Wong. Generating hardware from openmp programs. In *2006 IEEE International Conference on Field Programmable Technology*, pages 73–80, Dec 2006.

[76] P. Manet, D. Maufroid, L. Tosi, M. di Ciano, O. Mulertt, Y. Gabriel, J. D. Legat, D. Aulagnier, C. Gamrat, R. Liberati, and V. la Barba. Recops: Reconfiguring

programmable devices for military hardware electronics. In *2007 Design, Automation Test in Europe Conference Exhibition*, pages 1–6, 2007.

[77] Clive Maxfield. *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows*. Newnes, USA, 1st edition, 2004.

[78] Florian Mayer, Marius Knaust, and Michael Philippsen. Openmp on fpgas—a survey. In Xing Fan, Bronis R. de Supinski, Oliver Sinnen, and Nasser Giacaman, editors, *OpenMP: Conquering the Full Hardware Spectrum*, pages 94–108, Cham, 2019. Springer International Publishing.

[79] Hung Kiem Nguyen and Tu Xuan Tran. A survey on reconfigurable system-on-chips. *REV Journal on Electronics and Communications*, March 2018.

[80] K. O'Neal and P. Brisk. Predictive modeling for cpu, gpu, and fpga performance and power consumption: A survey. In *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 763–768, July 2018.

[81] David Padua. *Parallelization, Automatic*, pages 1442–1450. Springer US, Boston, MA, 2011.

[82] Peixin Zhong, P. Ashar, S. Malik, and M. Martonosi. Using reconfigurable computing techniques to accelerate problems in the cad domain: a case study with boolean satisfiability. In *Proceedings 1998 Design and Automation Conference. 35th DAC. (Cat. No.98CH36175)*, pages 194–199, 1998.

[83] A. Podobas. Accelerating parallel computations with openmp-driven system-on-chip generation for fpgas. In *2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs*, pages 149–156, Sep. 2014.

[84] Reid Porter, Jan Frigo, Al Conti, Neal Harvey, Garrett Kenyon, and Maya Gokhale. A reconfigurable computing framework for multi-scale cellular image processing. *Microprocessors and Microsystems*, 31(8):546–563, 2007. Special Issue on FPGA-based Reconfigurable Computing (3).

[85] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming heterogeneous systems from an image processing dsl. *ACM Trans. Archit. Code Optim.*, 14(3):26:1–26:25, August 2017.

[86] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *The Journal of Supercomputing*, 7(1-2):9–50, May 1993.

[87] Marc Reichenbach, Philipp Holzinger, Konrad Häublein, Tobias Lieske, Paul Blinzer, and Dietmar Fey. Heterogeneous computing utilizing fpgas. *Journal of Signal Processing Systems*, 91(7):745–757, Jul 2019.

[88] Gerald Roth, John Mellor-Crummey, Ken Kennedy, and R. Gregg Brickner. Compiling stencils in high performance fortran. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, SC '97, page 1–20, New York, NY, USA, 1997. Association for Computing Machinery.

[89] L. Sommer, J. Korinth, and A. Koch. Openmp device offloading to fpga accelerators. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 201–205, July 2017.

[90] M. Strickland. Fpga accelerated hpc and data analytics. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 21–21, Dec 2018.

[91] Russell Tessier and Wayne Burleson. *The Journal of VLSI Signal Processing*, 28(1/2):7–27, 2001.

[92] Miguel A. Vega-Rodríguez, Antonio Gómez-Iglesias, Juan A. Gómez-Pulido, and Juan M. Sánchez-Pérez. Reconfigurable computing system for image processing via the internet. *Microprocessors and Microsystems*, 31(8):498–515, 2007. Special Issue on FPGA-based Reconfigurable Computing (3).

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

[94] David W. Wall. Limits of instruction-level parallelism. *SIGARCH Comput. Archit. News*, 19(2):176–188, April 1991.

[95] Mirren White. The Adept Benchmark Suite, Jul 2019. [Online; accessed 15. Oct. 2019].

[96] Xilinx. Axi reference guide. `https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf`. (Accessed on 05/02/2021).

[97] Mohamed Zahran. Heterogeneous computing: Here to stay. *Queue*, 14(6):40:31–40:42, December 2016.

Appendix A

Software

This appendix shows in details the implementation of the system software. It is organized in a top-down perspective, from the Front-end Clang to the VC709 card driver. Section A.1 shows the modifications made to the front-end Clang, Section A.2 the modifications made to the OpenMP runtime, Section A.3 the modifications made to the libomptarget, and finally Section A.4 shows the changes performed on the card driver.

A.1 Clang Front-end

Clang is a compiler front-end for the C, C++, Objective-C and Objective-C++ programming languages. It uses the LLVM compiler infrastructure as its back-end and has been part of the LLVM release cycle since LLVM 2.6.

A.1.1 Fat Binaries Generation

As shown in [22] The code generation follows the following scheme:

1. For each source file provided, the compiler front-end driver spawns the execution of the pre-processor, compiler and assembler for the host and each available target device type. This results in the generation of an object file for each target device type.

2. Target linkers combine dedicated target objects into target shared libraries, one for each target device type.

3. The host linker combines host object files into an executable/shared library and incorporates each target shared libraries as is (no actual linking is done between host and target objects) into a designated section within the host binary. The format of a binary section for offloading to a specif device is target-dependent and will be thereafter handled by the target RTL at runtime.

4. a new driver command-line group option *fopenmp-targets=Ti*, where *Ti* is a valid target triples that specify which target device types the user wants to support in the execution of OpenMP target regions. All options following *fopenmp-targets=Ti*

are forwarded to that device toolchain. The user can specify as many *Ti* options as devices he/she wants to support. In our project, an example of invocation of the compiler would be as follows, for a hypothetical system where the available target device type is a VC709 board.

clang -fopenmp -fopenmp-targets=vc709-unknown-gnu-linux foo.c -o foo.bin

5. For each source file, the compiler driver will issue commands to create intermediate files for each possible compilation phase (LLVM IR, assembly, object) and target (host or device). However, this is not exposed to the user as the driver has the ability to bundle multiple files generated by different toolchains into a single one.

As for our work the code executed on the FPGA is hardware created in a separate flow, the front-end just replicates the x86_64 code generation to maintain compatibility with the entire compiler infrastructure. Therefore, the section in the fat binary reserved for the FPGA, is actually x86 code.

A.1.2 Code Generation

The clang front-end is also responsible for generating calls to the OpenMP runtime functions. One function that needed to be changed was the task creation function: *__kmpc_omp_target_task_alloc*. This change was focused in adding new parameters related to the device offloading, as shown in Listing A.1. The parameters are:

1. num_args: specifies the number of arguments that were passed in the map clauses
2. args_base: indicates the offset of the addresses of the arguments passed in the map () * clauses
3. args: indicates the base address of each argument passed in the map clauses;
4. args_sizes: indicates the size of each argument passed in the map clauses;
5. arg_types: indicates the type of each argument passed in the map clauses;
6. task_fname: indicates the name of the function that specifies the IP-core to be used which is extracted from the directive variant declare;

All this information is later added to the data structure that represents a task, as shown in Section A.2.1.

The resulting host executable/shared library will depend on the offload runtime library *libomptarget.so*. This library will handle the initialization of target RTLs and translate the offload interface from compiler-generated code to the target RTL during program execution. Details of libomptarget are shown in Section A.3.

```
KMP_EXPORT kmp_task_t *__kmpc_omp_target_task_alloc(
    ident_t *loc_ref,
    kmp_int32 gtid,
    kmp_int32 flags,
    size_t sizeof_kmp_task_t,
    size_t sizeof_shareds,
    kmp_routine_entry_t task_entry,
    kmp_int64 device_id,
    void *outlined_fn_id,
    kmp_int32 num_args,
    void **args_base,
    void **args,
    kmp_int64 *arg_sizes,
    kmp_int64 *arg_types,
    void *task_fname);
```

Listing A.1: __kmpc_omp_target_task_alloc.

```
struct kmp_taskdata {
  kmp_int32 td_task_id;
    /*other data.*/
  #ifdef USE_TARGET_TASK
    kmp_target_task_data_t *td_target_info; // Pointer to target nowait
    data info
   #endif
}; // struct kmp_taskdata
```

Listing A.2: kmp_taskdata structure.

A.2 OpenMP Runtime Modifications

The OpenMP runtime is a library of routines which help to manage a parallel program. It also includes routines to manage programs written in a task programming model.

At the time of publishing of this thesis, the LLVM/OpenMP implementation provided poor support to execute tasks on target devices. For this reason, it was necessary to make a few modifications.

The first modification made was in the data structure that represents a task. It was necessary to include data regarding the offloading operation. This data was encapsulated in a data structure called *kmp_target_task_data*, as shown in Listing A.3. This structure was inserted in the data structure of a task, *kmp_taskdata*, as shown in Listing A.2.

The fields in structure *kmp_target_task_data* are the same as those needed to be inserted into the task creation function *__kmpc_omp_target_task_alloc*. Details on the meaning of each field can be found in Section A.1.2.

A.2.1 Task Structure

Once this information was inserted into the task structure, it was necessary to create a communication interface between the runtime and the libomptarget offload library, so that libomptarget could know when and how to receive the task graph. Two functions were created for these purposes, and are, respectively, __kmpc_is_master_thread_tasking

```
1
2 struct kmp_target_task_data {
3   kmp_int64 device_id;  // Device ID
4   void *outlined_fn_id; // Target region/outlined function address
5   kmp_int32 num_args;   // Target region argument addresses and sizes
6   void **args_base;
7   void **args;
8   kmp_int64 *arg_sizes;
9   kmp_int64 *arg_types;
10  void *task_fname;
11 };
```

Listing A.3: kmp_target_task_data structure.

```
1 // Returns true if the master thread of the current team is tasking
2 // (thus if it is at a synchronization point).
3 KMP_EXPORT bool
4 __kmpc_is_master_thread_tasking ();
5
6 // Updates a task map with new tasks
7 KMP_EXPORT bool
8 __kmpc_target_task_map_update (kmp_target_task_map_t ** task_map);
```

Listing A.4: kmp_target_task_data structure.

and __kmpc_target_task_map_update, shown in Listing A.4.

the *__kmpc_is_master_thread_tasking* function returns false if the control thread has reached the end of a single region, that is, it has finished creating the tasks and the graph can now be obtained by *libomptarget*.

At this point, the function *__kmpc_target_task_map_update* is used by the libomptarget to obtain the graph. The function returns true if the graph is successfully obtained or false if there is a problem. The graph is obtained through the parameter *task_map*, which is a pointer to the *kmp_target_task_map_t* structure that, indirectly, represents the graph.

A.3 The Libomptarget Library

As previously said, the OpenMP 4.5 specification defines offloading directives that can be used to take advantage of accelerators devices. OpenMP terminology specifies three important concepts with respect to offloading, namely:

- **Device**: an implementation-defined logical execution unit. The execution model is host-centric such that a host device offloads code and data to target devices.
- **Target regions**: are structured code blocks that execute on a target device. This is conditional on the run-time availability of a device, the ability of the compiler to generate device code.

- **A mapped variable**: are variables in a (host) data environment with a corresponding variable in a device data environment.

The *libomptarget* is the library that provides the host with an API to map variables and initiate execution of target regions on a target device.

The target directive creates both a device data environment and a target region. It may have associated clauses to specify additional details, like the exact device to use if more than one is present in the system (device clause) or whether the data should be moved to/from the device or only allocated in the device memory (map clause).

The library utilizes device-specific target runtime libraries (RTLs). At the start of host code execution *libomtarget* will do the following:

1. Search for a target RTL that supports the device binary.
2. Verify target RTL interface compliance.
3. Add target RTLs into a list of available target device types.

After *libomptarget* has verified that suitable target code is present and that a target RTL is ready to execute a target region, the target RTL is invoked via API routines (described in Section A.3.1) to execute the region.

The offload library implements several compiler-level runtime library routines. Four are important for the execution in the FPGA cluster:

- **void __tgt_register_lib(__tgt_bin_desc *desc)**: registers the libomptarget.so library and initialize target state (i.e. global variables and target entry points) for the current host shared library/executable and the corresponding target execution images that have those entry points implemented. This does not trigger any execution in any target as any real work with the target device can be postponed until the first target region is encountered during execution. This function is expected to appear only once per host shared library/executable in the .init section and is called before any constructors or static initializers are called for the host.
- **int32_t tgt_target_data_begin(int32_t device_id , int32_t num_args, void** args_base, void** args, int64_t* args_size , int32_t *args_maptype)**: Initiates a device data environment. It maps variables from the host data environment to the device data environment by recording the mapping between the references of variables used in the host and target into the *libomptarget.so* internal structures. The associated variables in the target device data environment are initialized according to the map-type.
- **int32_t tgt_target_data_end(int32_t device_id , int32_t num_args, void** args_base, void** args, int64_t* args_size , int32_t *args_maptype)**: closes a device data environment. It removes mapped variables from the current device data environment, releases target memory and destroys the mappings created by *tgt_target_data_begin*(..) that initiated the current device environment. It assigns host variables with the value of the corresponding device data environment variable according to the map-type.

- **int32_t tgt_target(int32_t device_id , void *host_addr, int32_t num_args, void** args_base, void** args, int64_t* args_size , int32_t *args_maptype)**: Performs the same actions as **tgt_target_data_begin** in case **arg_num** is non-zero and launches the execution of the target region on the target device; if **arg_num** is non-zero after the region execution is done it also performs the same actions as **tgt_target_data_end**. If offloading fails, an error coe is returned, which notifies the caller that the associated target region has to be executed by the host. The return code can be used as an error code which will give the compiler and runtime the freedom to implement optimized behaviors.

The current implementation of this library can be classified into three components: target agnostic offloading, target specific offloading plugins, and target specific runtime library.

The target agnostic component is located in the file libomptarget.so and is the component that contains the logic to launch the initialization of the devices supported by the current program. It also creates device data environments and launches executions of kernels (OpenMP target regions). In order to deal with a specific device this component detects and loads the corresponding plugin. Details about plugins are in Section A.3.1.

A.3.1 The Plugins

They are loaded at runtime by libomptarget.so to interact with a given device. They all use the same interface and implement basic functionality like device initialization, data movement to/from device and kernel launching. Some of the functions that must be implemented are:

- **void __tgt_init_device()**: initializes the specified device. In case of success return 0; otherwise return an error code.

- **int32_t __tgt_rtl_load_binnary(int32_t device_id , __tgt_device_image *image)**: passes an executable image section described by image to the specified device and prepares an address table of target entities. In case of error, returns NULL. Otherwise, returns a pointer to the built address table. Individual entries in the table may also be NULL, when the corresponding offload region is not supported on the target device.

- **tgt_target_table* __tgt_rtl_is_valid_binary(__tgt_device_image* image)**: returns an integer different from zero if the provided device image can be supported by the runtime. The functionality is similar to comparing the result of tgt_rtl_load_binary to null. However, this is meant to be a lightweight query to determine if the RTL is suitable for an image without having to load the library, which can be expensive.

- **int32_t __tgt_rtl_data_alloc(int32_t device_id, int64_t size)**: allocates data on the particular target device of the specified size. Returns address of the allocated data on the target that will be used by libomptarget.so to initialize

the target data mapping structures. These addresses are used to generate a table of target variables to pass to __tgt_rtl_run_region(). This function returns NULL in case an error occurs on the target device.

- **int32_t __tgt_rtl_data_submit(int32_t device_id, void* target_ptr, void *host_ptr, int64_t size)**: passes the data content to the target device using the target address. In case of success, returns zero. Otherwise, returns an error code.

- **int32_t __tgt_rtl_data_retrieve(int32_t device_id, void* target_ptr, void *host_ptr, int64_t size)**: retrieves the data content from the target device using its address. In case of success, returns zero. Otherwise, returns an error code.

- **int32_t __tgt_rtl_data_delete(int32_t device_id, void* target_ptr)**: deallocate the data referenced by target_ptr on the device. In case of success, returns zero. Otherwise, returns an error code.

- **int32_t tgt_target(int32_t device_id , void *host_addr, int32_t num_args, void** args_base, void** args, int64_t* args_size , int32_t *args_maptype)**: Performs the same actions as **tgt_target_data_begin** in case **arg_num** is non-zero and launches the execution of the target region on the target device; if **arg_num** is non-zero after the region execution is done it also performs the same actions as **tgt_target_data_end**. If offloading fails, an error code is returned, which notifies the caller that the associated target region has to be executed by the host. The return code can be used as an error code which will give the compiler and runtime the freedom to implement optimized behaviors.

Each plugin must interact with your device by taking the necessary actions to implement these functions. In our case, the plugin interacts directly with the VC709 board driver, as shown in Section A.4.

A.4 Driver Organization

Figure A.1 shows the stack of blocks that composes the VC709 board driver. The V709 plugin was inserted at the top of the stack, in the user space, to use the resources of the drive and consequently the card.

The Kernel Space is composed by the User Driver and the DMA Driver (see Figure A.1) and provides the DMA engine configuration required to achieve data transfer between the hardware and the main system memory. The data transmission works as follows: (a) on the transmit path, data from the application is handed over to the driver for transmission. The driver then queues up the packet for scatter-gather DMA in the FPGA. The DMA fetches the packet through the PCIe Endpoint and transfers it to the XGEMAC where it is transmitted through the Ethernet link into the LAN; (b) on the receive side, packets arriving on XGEMAC are collected by the scatter-gather DMA. The DMA pushes the packet to the driver through the PCIe Endpoint. The driver hands off the packet to the upper layers for further processing.

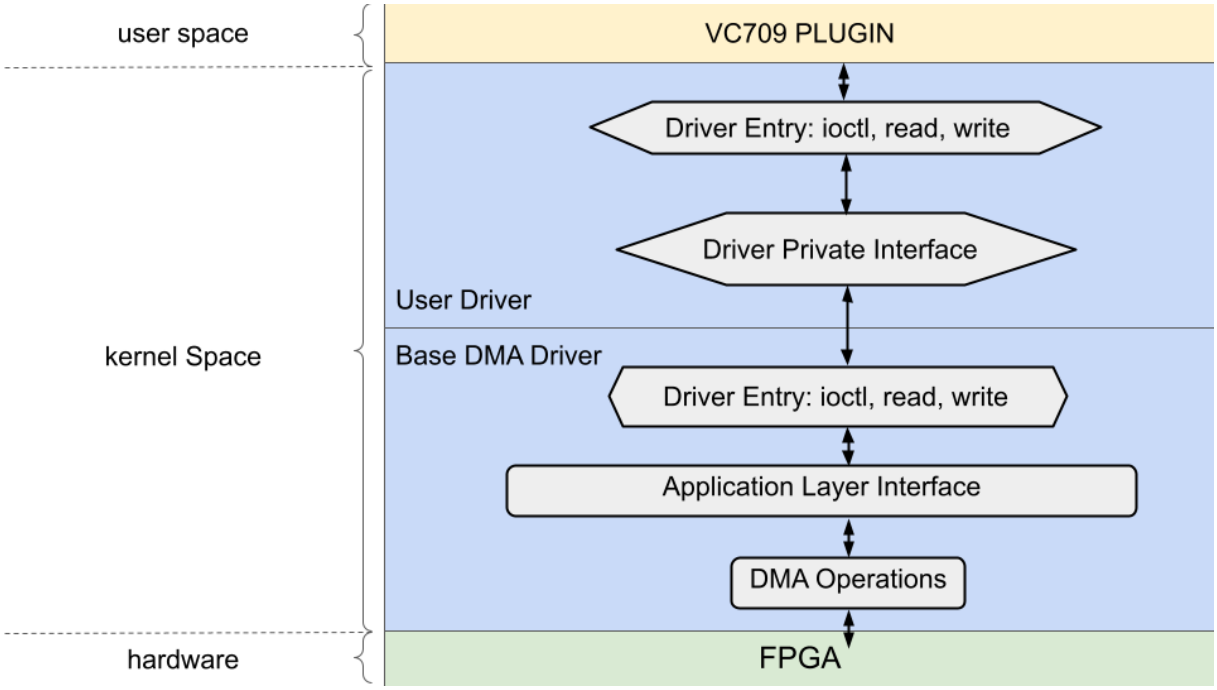


Figure A.1: Driver organization.

Both User Driver and Base DMA driver have a driver entry block with three main functions (ioctl, read and write) (Figure A.1). The *read* and *write* functions are used for data connectivity and the *ioctl* function is used for configurations. Standard network tools use driver entry points for Ethernet configurations. The driver hooks in entry points configure 10G Ethernet MAC and PHY. The other driver entry points are mainly used in the data flow for transmitting and receiving Ethernet packets.

The read and write functions are used to exchange data with the FPGA. Since execution on the FPGA is performed in stream mode, the read function must be called before the write function, so that the data receiving infrastructure is ready when the FPGA starts to return the computed data. The ioctl function is used for control signals and it uses the data structure shown in Listing A.5. The Engine, *TestMode*, *MinPktSize* and *MaxPktSize* fields are original to the driver and are used for testing purposes. The *address* and *value* fields, on the other hand, needed to be added to make it possible to configure the modules inserted in hardware. The *address* and *value* fields indicate the register to be configured and the value, respectively. Listing A.6 shows the code that was inserted in the driver to handle the IOCTL calls regarding the configuration of the registers inserted in the architecture. Listing A.7 shows an example of how the plugin writes to one of the registers.

The driver private interface (Figure A.1) enables interaction with the DMA driver through a private data structure interface. The data that comes from the user application through the driver entry points is sent to the DMA driver through the private driver interface. The private interface handles received data and housekeeping of the completed transmit and receive buffers by putting them in a completed queue.

The Application Layer interface is the block responsible for dynamic registering and unregistering the user application drivers. The data that is transmitted from the user

```
typedef struct {
    int Engine;              /* Engine Number */
    unsigned int TestMode;   /* Test Mode - Enable TX, Enable loopback */
    unsigned int MinPktSize; /* Min packet size */
    unsigned int MaxPktSize; /* Max packet size */
    int address;            /* used to address configuration registers */
    int value;              /* used to set up configuration registers.*/
} TestCmd;
```

Listing A.5: OpenMP tasks in CPUs.

```
xraw_dev_ioctl (struct file *filp, unsigned int cmd, unsigned long arg){
  switch (cmd){
    case ISTART_TEST:{
      u32 addrs = TXbarbase + (u32)((TestCmd *)arg)->address;
      int value = (TestCmd*)arg)->value
      XIo_Out32 (addrs, value);
      break;
    }
  }
}
```

Listing A.6: IOCTL for configuration.

```
  TestCmd cmd;
  cmd.address = 0x9470;
  cmd.value = conf;
  ret_val = ioctl(EngineFD, ISTART_TEST, &cmd);
```

Listing A.7: DMA configuration example.

application driver is sent over to the DMA operations block.

User application driver sends the received socket buffer packet to DMA driver for mapping to PCI space and sending it to DMA. On the receiver side buffers are pre-allocated to store incoming packets. These packets are allocated from networking stack. The received packets are added to network stack queue for sending it to application for further processing.

The DMA Operations block works as follow: for each DMA channel, the driver sets up a buffer descriptor ring. At the beginning of execution, the receive ring (associated with a C2S channel) is fully populated with buffers meant to store incoming packets, and the entire receive ring is submitted for DMA while the transmit ring (associated with a S2C channel) is empty. As packets arrive at the base DMA driver for transmission, they are added to the buffer descriptor ring and submitted for DMA transfer.

Appendix B

Hardware

This appendix shows in details the implementation of the system hardware. It is organized in the host-fiber sense, that is, starting from the communication with the host machine until the communication with the optical fiber, covering the main modules used, namely: (a) Section B.1 shows the PCIE and DMA IP modules; (b) Section B.2 shows the Configuration Registers; (c) Section B.3 shows the A-SWT module; (d) Section B.4 shows the IP-cores implementation; (f) Section B.5 shows the MFH module; (g) Section B.6 shows the VFIFO module; and (h) Section B.7 shows the Network Subsystem modules.

B.1 PCIe and DMA

The PCIe standard is a high-speed serial protocol that allows transfer of data between host system memory and Endpoint cards. To efficiently use the processor bandwidth a bus-mastering scatter-gather DMA controller is used to push and pull data from the system memory.

The Virtex XT FPGA Integrated Block for PCIe provides a wrapper around the integrated block in the FPGA. The XT Connectivity TRD uses PCIe in x8 Gen3 configuration and buffering set for high performance applications. Figure B.1 shows a block design for the Virtex XT FPGA Integrated Block for PCIe. The ports can be organized as follows:

- The ports *m_axis_cq_**, *s_axis_cc_**, *s_axis_rq_**, *m_axis_rc_** are used for AXI stream communication between the PCIe module and the DMA;
- clock and reset interface are used for synchronization;
- *pci_exp** are used to connect with the physical pins of the PCI express interface;
- *configuration and status ports* they are all other ports, are used for configuration and status reading.

The Scatter Gather Packet DMA IP (DMA IP) is provided by Northwest Logic, Listing B.1 shows the Verilog IP simplified interface. The front-end of the DMA interfaces with the AXI4-Stream interface on PCIe Endpoint IP core (ports s_axis_rq_t*, m_axis_rc_t*, m_axis_cq_t*, s_axis_cc_t*, fc_* and cfg_* in Listing B.1). The backend of

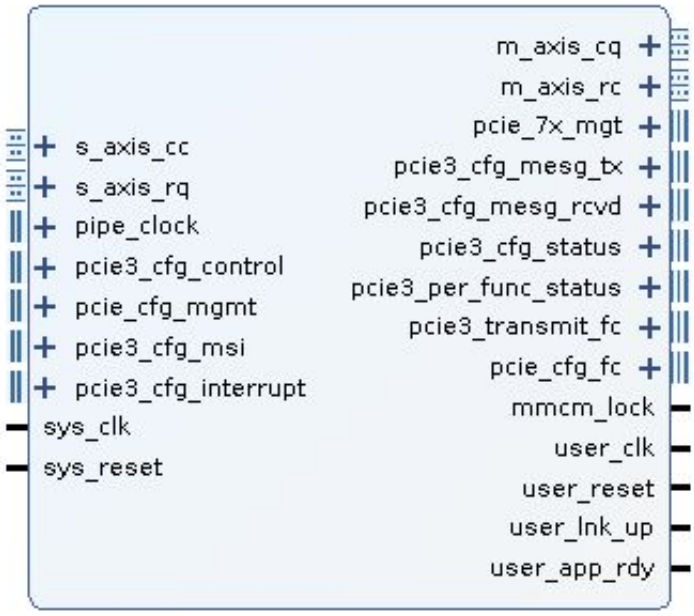


Figure B.1: Integrated Block for PCI Express.

the DMA provides an AXI4-Stream interface as well, which connects to the user application side. This DMA controller is configured to support simultaneous operation of four user applications using eight channels in all. This involves four system-to-card (S2C) or transmit channels and four card-to-system (C2S) or receive channels (ports s2c* and c2c* in Listing B.1). The s2c* and c2c* channels implement all the main ports of the AXI4-STREAM protocol: tdata, tlast, tvalid, tready, tkeep, tstrob and tuser.

The term scatter gather refers to the ability to write packet data segments into different memory locations and gather data segments from different memory locations to build a packet. This allows for efficient memory utilization because a packet does not need to be stored in physically contiguous locations. Scatter gather requires a common memory resident data structure that holds the list of DMA operations to be performed. DMA operations are organized as a linked list of buffer descriptors. A buffer descriptor describes a data buffer.

The DMA IP also has an AXI Lite interface (ports t* in Listing B.1) that is used to configure modules within the FPGA. Section B.2 shows how this interface is used to access *configuration registers*.

B.2 Configuration Registers

Configuration Registers are a set of registers used to configure some internal FPGA modules. They are addressed by the DMA IP AXI Lite interface. Address offsets from 0x0000 to 0x7FFF are consumed internally by the DMA engine. Address offset space from 0x8000 to 0xFFFF is provided to user. Transactions targeting this address range are made available on the DMA AXI4 Lite interface.

The design uses the range 0x9000 to 0x9FFF to implement the required user space

```
module packet_dma_axi
(

    input                                   user_reset,
    input                                   user_clk,

    output                                  s_axis_rq_t*
    input  [CORE_DATA_WIDTH-1:0]            m_axis_rc_t*
    input  [CORE_DATA_WIDTH-1:0]            m_axis_cq_t*
    output [CORE_DATA_WIDTH-1:0]            s_axis_cc_t*
    inout  [11:0]                           fc_*

    // PCIe Configuration Interface
    inout                                  cfg_*,

    //DMA - C2S Engine #0
    input                                   c2s0_*,
    //DMA - C2S Engine #1
    input                                   c2s1_*,
    //DMA - C2S Engine #2
    input                                   c2s2_*,
    //DMA - C2S Engine #3
    input                                   c2s3_*,

    //DMA - S2C Engine #0
    output                                  s2c0_*,
    //DMA - S2C Engine #1
    output                                  s2c1_*,
    //DMA - S2C Engine #2
    output                                  s2c2_*,
    //DMA - S2C Engine #3
    output                                  s2c3_*,

    //DMA - AXI lite
    inout                                   t_*
);
```

Listing B.1: DMA IP Verilog Simplified Interface.

```
module itt_ctrl (
  //General signals:
  input wire ACLK,
  input wire RSTN,
  //Slave AXIS interface:

  //Slave AXIS interface:
  input wire [255:0]S_AXIS_T*,

  //Master AXIS interface:
  output wire [255:0]M_AXIS_T*

  input wire input_conf,
  input wire [31:0] input_size,

    input wire [3:0] init_swt,
  input wire [3:0] init_sup,
  input wire [3:0] conf_swt,
  input wire [3:0] conf_sup,
  output reg [3:0] output_swt,
  output reg [3:0] output_sup
);
```

Listing B.2: Iteration Control IP Simplified Interface.

registers; 0xB000 to 0xEFFF to implement the address space four MACs. Table B.1 shows the configuration registers that are used in the project.

The architecture also uses the 1x5 AXI4LITE Interconnect to route the request to the right slave.

B.3 A-SWT

The A-SWT is a hierarchy of switches that allows communication between the IP-cores within the FPGA. Each switch is a AXI4-Stream Interconnect that enables the connection of heterogeneous master/slave AXI4-Stream protocol compliant endpoint IP. The AXI4-Stream Interconnect routes connections from one or more AXI4-Stream master channels to one or more AXI4-Stream slave channels. Figure B.2 shows the interface of module Integrated Block for AXI-Stream Interconnect RTL which is used as the basic block for the construction of module A-SWT. The module in question has an input AXI-Stream interface, S00_AXIS, and two output AXI-Stream interfaces, M00_AXIS and M01_-AXIS.

The A-SWT module also has a sub-module called itt_ctrl. This sub-module is responsible for configuring the reuse of the IP-cores. This reuse is done using the switch's supports according to the number of iterations to be calculated. Listing B.2 shows the simplified Verilog interface of the itt_ctrl submodule. The module receives as input the initial values of the switches and supp ports, the values to be configured, the number of iterations to be computed, and the dimensions of the grid. With these values, the sub-module is able to modify the configuration values at run time.

Table B.1: Configuration Registers

REGISTER	ADDRESS	DESCRIPTION
DST_MAC0_ADRS_FILTER	0x9430	Registers referring to the MAC addresses of the fiber optic interfaces.
DST_MAC0_ADRS_LOW	0x9434	
DST_MAC0_ADRS_HIGH	0x9438	
DST_MAC1_ADRS_FILTER	0x943C	
DST_MAC1_ADRS_LOW	0x9440	
DST_MAC1_ADRS_HIGH	0x9444	
DST_MAC2_ADRS_FILTER	0x9448	
DST_MAC2_ADRS_LOW	0x944C	
DST_MAC2_ADRS_HIGH	0x9450	
DST_MAC3_ADRS_FILTER	0x9454	
DST_MAC3_ADRS_LOW	0x9458	
DST_MAC3_ADRS_HIGH	0x945C	
DST_SIZE0	0x9460	Registers referring to the sizes of the data to be transported on the fiber interfaces.
DST_SIZE1	0x9464	
DST_SIZE2	0x9468	
DST_SIZE3	0x946C	
SWT_ADDRS	0x9470	Switch initial value
MTUS_SIZE	0x9474	MTU size
CORE_CONF	0x9478	Set Configuration
SUPPRESS_INIT_ADDR0	0x947C	Initial suppression value of IP-core 0. Used for the reuse of IP-cores.
SUPPRESS_INIT_ADDR3	0x9480	Initial suppression value of IP-core 3. Used for the reuse of IP-cores.
SUPPRESS_CONF_ADDR0	0x9484	IP-core 0 suppression configured value. Used for IP-core reuse
SUPPRESS_CONF_ADDR3	0x9488	IP-core 0 suppression configured value. Used for IP-core reuse
SWT_INIT_ADDR0	0x948C	initial value of the IP-core 0 switch.
SWT_INIT_ADDR3	0x9490	initial value of the IP-core 3 switch.
SWT_CONF_ADDR0	0x9494	configured value of the IP-core 0 switch.
SWT_CONF_ADDR3	0x9498	configured value of the IP-core 3 switch.
GRID_LENGTH_ADDR	0x949C	Grid length
GRID_WIDTH_ADDR	0x94A0	Grid width
GRID_HEIGHT_ADDR	0x94A4	Grid height
ITERATION_CONF	0x94A8	Number of iterations to be calculated.

```
module ipStencil(
  //General signals:
  input wire ACLK,
  input wire RSTN,
  output wire Error,

  //Slave AXIS interface:
  input wire [Block_Width*`Precision-1:0]S_AXIS_T*,

  //Master AXIS interface:
  output wire [Block_Width*`Precision-1:0]M_AXIS_T*,

  input wire [31:0] input_length,
  input wire [31:0] input_width,
  input wire [31:0] input_height
);
```

Listing B.3: IP interface.

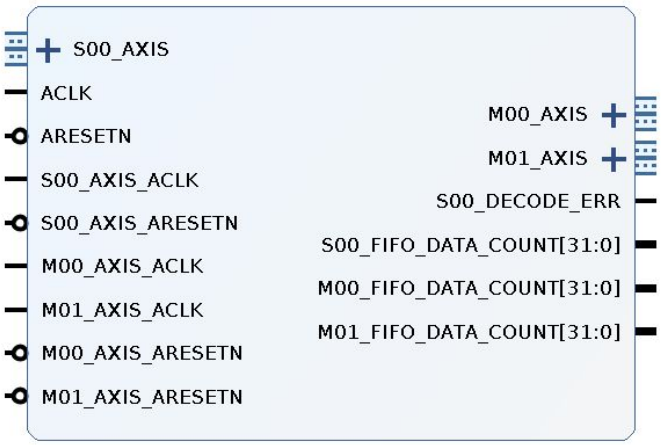


Figure B.2: Integrated Block for AXI-Stream Interconnect RTL.

B.4 The IP-cores

The IP-cores that were designed work in stream mode. For this reason, they communicate with the infrastructure using an AXI-Stream interface. Listing B.3 shows the simplified Verilog code interface for an IP-core. Ports S_AXIS_T* is an AXI-Stream Slave interface, that is, it is the interface through which IP-core receives data. The M_AXIS_T* ports are the AXI-Stream master interface through which the IP-core transmits the data it produces. The input_length, input_width and input_height ports specify the dimensions of the grid that the IP-core computes. Its maximum dimensions are fixed at synthesis time, but the real limits may be set on at run time.

```
module add_header(
  input wire ap_clk,
  input wire ap_rst_n,
  input wire ap_start,

  input wire inStream_T*,
  output reg outStream_T*

  input wire [31 : 0] mtus_V,
  input wire [31 : 0] remainder_V,
  input wire [47 : 0] dst_V,
  input wire [47 : 0] src_V
);
```

Listing B.4: MFH add header module.

```
module add_header(
  input wire ap_clk,
  input wire ap_rst_n,
  input wire ap_start,

  input wire inStream_T*,
  output reg outStream_T*

  input wire [31 : 0] mtus_V,
  input wire [31 : 0] size_V
);
```

Listing B.5: MFH remove header module.

B.5 MFH

The MFH Module is responsible for adding and removing the MAC header of the frames that are sent and received through the fiber optic link. Listing B.4 shows the simplified interface of the add_header submodule. The inStream_T * and outStream_T * ports are AXI Stream ports, all signals are not shown for simplicity. The mtus_V and remainder_V ports indicate the size of the data that will be transmitted. The mtus_V port, precisely informs the count of complete MTUs that will be transmitted, while the remainder_V port what is left, that is, has not completed an MTU.

The Equation B.1 shows how the mtus_V is calculated, and Equation B.2 shows how the remainder_V is calculated.

$$mtus\_V = TOTAL/MTU\_SIZE \tag{B.1}$$

$$remainder\_V = TOTAL\%MTU\_SIZE \tag{B.2}$$

Listing B.5 shows the simplified interface of the rm_header sub-module. The ports are similar to those of the add_header sub-module, with the exception that it has no source and destination, since this module removes the header.

```
1
2  module axis_vfifo_ctrl_ip #(
3    )(
4          //- SODIMM-1
5      output [15:0]                                c0_ddr_addr,
6      output [2:0]                                 c0_ddr_ba,
7
8          //- SODIMM-2
9      output [15:0]                                c1_ddr_addr,
10     output [2:0]                                 c1_ddr_ba,
11    // AXI streaming Interface for Write port
12    input  [NUM_PORTS-1:0]                        axi_str_wr_t*
13
14    // AXI streaming Interface for Read port
15    output [NUM_PORTS-1:0]                        axi_str_rd_t*,
16
17 );
```

Listing B.6: VFIFO interface.

B.6 VFIFO

The XT Connectivity TRD uses DDR3 space as multiple FIFOs for storage. Listing B.6 shows a simplified version of the Verilog interface for this module. The c0* and c1* ports are the ports that connect with the physical pins of the two memory chips. Whereas the *axi_str_wr_t** and *axi_str_rd_t** ports are AXI4-Stream interfaces that the module provides so that other internal FPGA modules can read and write in the memory chips.

B.7 Network Subsystem

The TRD uses four modules to communicate with the four fiber interfaces. Each of these modules are called net_ip and the four together form the Network Subsystem. Listing B.7 shows a simplified version of the Verilog interface of a net_ip module.

The xphy* ports are the ports that connect with the physical pins of the fiber optic interface, whereas the *axi_str_wr_t** and *axi_str_rd_t** ports are AXI4-Stream interfaces that the module provides so that other internal FPGA modules can read and write in the fiber optic interfaces. Finally, the s_axi_l is a AXI4-Lite interface used for configuration.

```
module net_ip (
    // AXI Lite register interface
    input                               s_axi_l*,
    // AXI Streaming data interface
    input  [AXIS_TDATA_WIDTH-1:0]       axi_str_wr_t*,

    output [AXIS_TDATA_WIDTH-1:0]       axi_str_rd_t*,

`ifdef USE_XPHY
    input                               xphy_refclk_n,
    input                               xphy_refclk_p,
    output                              xphy_txp,
    output                              xphy_txn,
    input                               xphy_rxp,
    input                               xphy_rxn
);
```

Listing B.7: Simplified NET interface.