DESIGN AND IMPLEMENTATION OF AN ADAPTIVE HIGH EFFICIENCY FPGA-ACCELERATED SYSTEM FOR MULTI-DISCIPLINARY APPLICATION DOMAINS

by

SVORONOS LEIVADAROS

B.Sc., Technological Educational Institute of Crete, 2017

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

SCHOOL OF ENGINEERING

HELLENIC MEDITERRANEAN UNIVERSITY

2020

Approved by:

George Kornaros

Abstract

The scope of this thesis is the design and implementation of an FPGA platform that utilizes state-of-the-art techniques and methodologies to allow improved energy efficiency and performance in carrying out computationally intensive tasks. The goal is to develop a framework for FPGA-based architectures that can be used in environments that include but are not limited to Cloud Computing Clusters, High Performance Computational Clusters and Distributed Data Centers. A proof of concept implementation of this framework with 3 accelerated tasks (Black and White Image Thresholding, Image Convolution with a 3x3 Kernel, Genome 2-mer Distribution Analysis) is also presented and compared with implementation of the same tasks on an FPGA and ARMv7 architecture CPU. To our knowledge, the methodology in designing the Partial Reconfiguration platform employed in this work is novel and allows designing Dynamic Partial Reconfiguration-enabled hardware platforms on an FPGA without the need for wrapper logic or the need to register all inputs and outputs to every reconfigurable module, facilitating the establishment of compatibility across implemented reconfigurable modules in early development and future integration of new accelerated functions on the FPGA platform. Performance and energy efficiency metrics are also presented for the 2 different implementation platforms.

Περίληψη

Το αντικείμενο μελέτης αυτής της διπλωματικής εργασίας είναι η σχεδίαση και υλοποίηση μιας πλατφόρμας Συστοιχίες Πυλών Προγραμματιζόμενες στο Πεδίο (FPGA) η οποία αξιοποιεί σύγχρονες τεχνικές και μεθοδολογίες για να επιτύχει μέγιστη ενεργειακή αποδοτικότητα και απόδοση στην διεκπεραίωση υπολογιστικά απαιτητικών διεργασιών ευρείου φάσματος. Ο στόχος είναι να αναπτύξουμε ένα πλαίσιο λογισμικού για υλοποίηση αρχιτεκτονικών υπολογιστικών συστημάτων βασισμένα σε FPGAs για χρήση σε περιβάλλοντα εργασίας όπως Συστοιχίες Υπολογιστικών νεφών (Cloud Computing Clusters), Υπολογιστικές Συστοιχίες Υψηλής Απόδοσης (High Performance Computing) και Κατανεμημένα Κέντρα Δεδομένων (Distributed Data Centers). Μια υλοποίηση proof-of-concept του προτεινόμενου πλαισίου με 3 εφαρμογές που μπορούν να επιταχυνθούν (Ασπρόμαυρη Κατωφλίωση Εικόνας, Συνέλιξη εικόνας με πυρήνα διαστάσεων 3x3, Κατανομή Διμερών Νουκλεοτιδίων Γονιδιωμάτων) θα παρουσιασθεί και θα συγκριθεί με αντίστοιχες υλοποιήσεις σε συμβατικές αρχιτεκτονικές υπολογιστών με επεξεργαστή ARMv7. Κατά την εκτίμηση μας, αυτή είναι η πρώτη έρευνα που περιγράφει μια νέα μεθοδολογία που να επιτρέπει τον σχεδιασμό πλατφόρμων με δυνατότητες Δυναμικής Μερικής Αναδιαμόρφωσης χωρίς την ανάγκη υλοποίησης λογικής περιτύλιξης ή κατοχύρωσης όλων των εισόδων και εξόδων σε όλες τις αναπροσαρμοζόμενες μονάδες, διευκολύνοντας την εγκαθίδρυση συμβατότητας μεταξύ των υλοποιημένων μονάδων επιτάχυνσης και την μελλοντική επέκταση της πλατφόρμας με νέους αλγορίθμους, επιταχυνόμενους από την πλατφόρμα FPGA. Επίσης, θα παρουσιάσουμε μετρήσεις επιδόσεων και ενεργειακής απόδοσης από τις υλοποιημένες πλατφόρμες.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

The fields of Computer Engineering and Computer Science have played a significant role in the advancement of nearly every aspect of human society. Computational systems have found use in multiple disciplines with the result of facilitating experts in all scientific and engineering fields to advance their respective field's knowledge and increase our quality of life.

In recent years, a field of computational systems based on clusters of interconnected processing and storage nodes that are connected to the internet has been established as the go-to method for covering a wide range of computational and data storage needs. This field is called Cloud Computing and many variations of Cloud Computing architectures and models have arisen as a result of extended research on the topic.

Although popularized in 2006 with the release of the AWS (Amazon Web Services) platform, the term 'cloud computing' is believed to have been first coined in 1996 [1] by Sean O'Sullivan in a business plan report detailing the need for a migration of communication and collaboration systems to the 'Internet cloud'.

Cloud computing solutions and ongoing research focus on a wide range of objectives which includes but is not limited to the following:

- Secure remote storage of sensitive data (Microsoft OneDrive, Google Drive, Dropbox) [2]
- Mass processing of data acquired from web services and IoT Devices [3]
- Big Data and Machine Learning algorithms to extract trends in demand of services and products [4]
- An application development platform where projects are shared, stored and compiled on the computing cluster instead of the user's personal computer [5]
- Efficient acceleration of data intensive processes thanks to the economies of scale [6]

Another field of Computer Science deals with development of computational systems on specialized integrated circuits called FPGAs (Field-Programmable Gate Arrays). FPGAs are meshes of primitive logic cells that usually consist of look-up tables, flip-flops and digital signal processors. Both the primitive blocks of logic as well as the interconnections in the fabric can be programmed and configured as per the developer's needs.

The designs on an FPGA fabric are usually developed using a special type of programming language called Hardware Description Language such as VHDL and Verilog to selectively interconnect these primitive logic cells in such a way that they perform a specific task or even a whole algorithm.

Another type of programming paradigm used to design FPGA hardware platforms is High-Level Synthesis (HLS). HLS refers to the methodology where a developer uses a more conventional programming language more often used in developing applications in static, mainstream CPU architectures such as C, C++ or SystemC. The resulting code is then parsed and transformed into equivalent HDL code (usually VHDL or Verilog). HLS as a development methodology is advertised as an enabler of shorter development cycles, reduced time-to-market and facilitation of porting implemented C code to new devices.

The complexity of such designs can range from basic mathematical and logic operations such as addition and comparison to fully implemented algorithms that deal with data encryption [7], image filtering [8], video processing [9] and many more.

Some of the benefits of implementing such designs on an FPGA compared to other platforms such as a CPU or a GPU include but are not limited to the following:

- Considerable speedup compared to CPU implementations
- Comparable performance to GPU accelerated implementations
- Considerable increase in performance per watt compared to CPU and FPGA implementations
- Decreased overall system power consumption
- Soft and hard real-time application capabilities

It is the aim of this thesis to explore the viability of utilizing FPGA-based systems in Cloud Computing environments that leverage multidisciplinary workloads.

The goal is to develop a cloud computing platform that performs computationally intensive tasks by utilizing algorithms implemented on an FPGA. FPGAs can offer lots of benefits to businesses and organizations if utilized correctly and to their full capabilities.

There are several objectives that the proposed solution aims to cover. The 3 main characteristics of the proposed system are

1. **Adaptability:** the proposed system is capable of adapting to a computing environment where data-intensive acceleration requests for a wide variety of functions take place. In the proposed system, this adaptability refers to the ability to efficiently reprogram portions of the FPGA marked as reprogrammable in a manner that minimizes the chance that a reprogramming is needed by employing a number of techniques which includes but is not limited to the following

a. Utilizing reconfigurable modules that are already programmed in the FPGA fabric
b. If reprogramming is needed, reprogram Reconfigurable Partitions on the FPGA that are least recently used.

2. **High Efficiency:** the proposed FPGA-based platform must implement techniques and design methodologies that will allow it to offer the maximum possible energy efficiency, while also maintaining high performance and QoR (Quality of Results) comparable to Cloud Computing platforms based on other, non-FPGA based architectures. Dynamic Partial Reconfiguration is the main tool driving this objective. Additionally, reprogramming a RP (Reconfigurable Partition) with a blank bitstream after it has not been used for some time can also help with decreasing idle power consumption of the system.

3. **Multi-disciplinary Task Execution:** the proposed system is designed in a way that accommodates the execution of many different algorithms on the same Reconfigurable Partitions (RP). Any type of algorithm that might work on different types of data such as image data or genome sequences can be programmed in the defined RPs and used by a system user, as long as specific conventions, outlined in later chapters, have been followed at design time.

Several methodologies and techniques, some specific to FPGAs, will be employed in order to ensure that the developed platform meets power efficiency, performance and QoR metrics that Cloud Computing solutions can benefit from. These include but are not limited to the following

- Dynamic Partial Reconfiguration (DPR)
- Workload parallelization
- Custom Pipelining Architectures
- Energy-aware scheduling

In the next chapters the following will be covered:

- In chapter 2 a theoretical background will be presented that pertains to the methodologies, techniques and systems employed in this work as well as related work on the field of Cloud Computing, FPGAs and Energy Efficient Computing.
- In chapter 3 the architecture of the proposed system will be presented along with the software and hardware development environments utilized as well as the hardware specifications of the target FPGA platform.
- In chapter 4 implementation details regarding the proposed system as well as pseudo code and algorithms that were utilized as use cases will be outlined.
- In chapter 5 performance metrics such as computational throughput and energy consumption of the implemented will be presented.
- In chapter 6, we discuss and comment on the system's operational results.
- Finally in chapter 7 we conclude this report and discuss future work that can extend the findings of this thesis.

2. Theoretical Background and Related Work

In this chapter, information relating to the theories, techniques and technologies employed and build upon in this thesis are presented. Additionally, related research work on Cloud Computing and FPGAs, both as separate fields as well as in conjunction with each other, will be presented.

2.1. Cloud Computing Definition

Businesses and organizations constantly strive to meet their strategic goals and objectives by minimizing their operational costs and increasing the quality of services and products they offer.

The field of Cloud Computing has facilitated the operation of organizations by presenting an opportunity to offload data processing and storage to clusters of compute nodes and storage facilities and use the processing power offered by these clusters to employ Machine Learning and Big Data Analytics applications that can help shape the strategic choices of businesses.

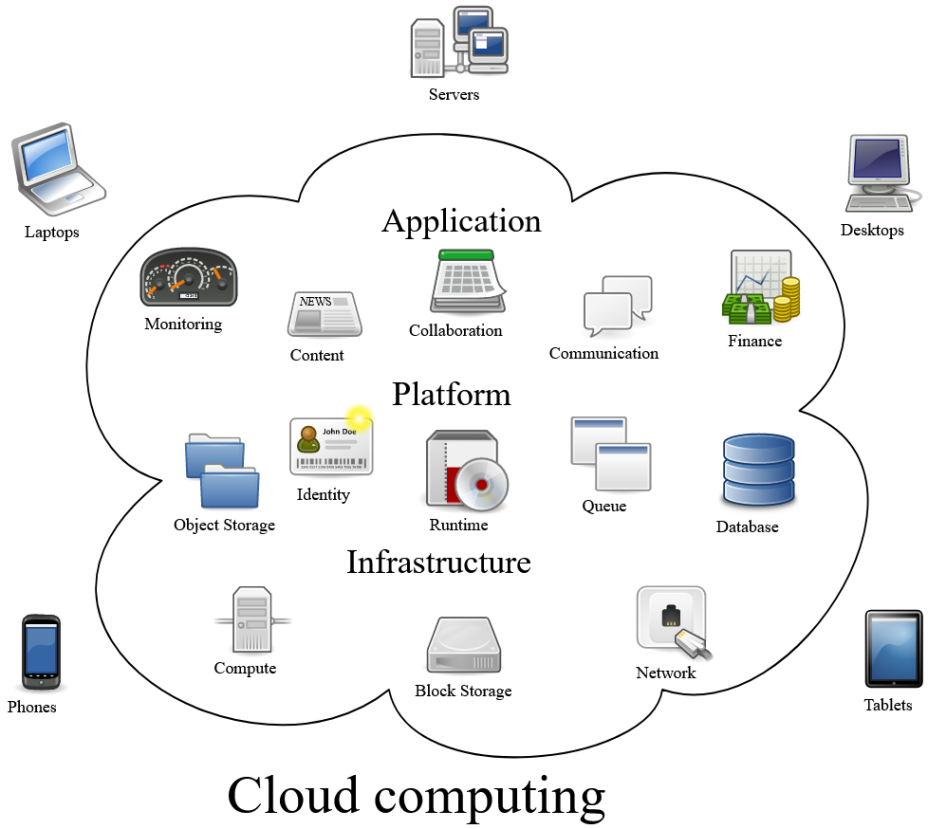


Figure 1 – Architectural Diagram of Cloud Computing - Created by Sam Johnston using OmniGroup's OmniGraffle and Inkscape https://commons.wikimedia.org/w/index.php?curid=6080417

But what exactly is Cloud Computing? As with many term definitions, the linguistic definition of Cloud Computing is subject to personal interpretation. Several definitions have been given by experts and researchers on the field.

The National Institute of Standards and Technology (NIST) [10] defines cloud computing as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction".

In 2009, in their research on current trends and design considerations regarding Cloud Computing integration on organizational and business environments with the goal of establishing Computing as the 5th utility, Buyya et. al [11] described Cloud Computing as "a type of parallel and distributed system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resource(s) based on service-level agreements established through negotiation between the service provider and consumers."

2.2. Fundamental Characteristics of Cloud Computing

When deploying a cloud compute cluster, there are specific requirements that such a platform should meet in order to fully realize its goal of efficiently carrying out the processing and storage of massive amounts of data from different users.

Correspondingly, in 2011 NIST [10] included in its definition of Cloud Computing 5 essential characteristics that define a well-implemented cloud computing cluster. These characteristics are

1. **On-demand self-service**: An end-user can utilize computing resources such as server time and storage automatically without the need for human interaction with service providers.
2. **Broad network access**: Cloud capabilities are readily available over internet connection enabled devices and accessed through standard mechanisms that allow usage using a variety of client platforms such as personal computers, mobile phones and tablets.
3. **Resource pooling**: The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, and network bandwidth.
4. **Rapid elasticity**: Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.

5. **Measured service**: Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

2.3. Field Programmable Gate Arrays (FPGAs)

Field-Programmable Gate Arrays are specialized integrated circuits that are structured in such a way that allows them to be electronically reprogrammed, changing their functionality according to users' needs on the fly.

An FPGA is comprised of various types of primitive programmable logic and reconfigurable wiring that allows the fabric inside to be connected in a way specified by a Hardware Description Language (HDL) in order to carry out either a simple task such as an AND or XOR calculation or a complex task or algorithm such as a Sobel Image Filter or an encryption scheme like AES.

Besides specialized task acceleration, FPGAs can be programmed to operate like a CPU that implements a custom or standardized architecture of instructions like RISC. These types of processors are usually referred to as 'Soft cores' and many parameters such as pipeline depth or cache size can be user-defined. An example of a Soft Core is MicroBlaze [12].

FPGAs as the name suggests can be "programmed on the field" after deployment. This task can be carried out by the device itself which according to operational needs can reprogram the FPGA with new bitstreams, implementing new functionality. This type of device is referred to as In-System Programmable (ISP) [13].

An FPGA-based implementation of a task is usually much faster and more efficient than similar implementations on x86_64 or ARM architecture CPUs such as the ARM A9 or Intel CPUs [14], [15]. This benefit is usually offset by that fact that HDL-based designs are harder to develop, debug and test than implementations of algorithms running on conventional CPU architectures, although extensive effort to facilitate development of FPGA designs has been made in the last years through the use of High-Level Synthesis development paradigms.

FPGAs offer many benefits when compared to ASICs such as design reuse and ease of maintenance. Errors made in the design process can be easily fixed after provisioning an FPGA device by sending the new corrected bitstreams. This is not possible for an ASIC platform. Additionally, ASICs incur very high non-recurrent engineering costs compared to FPGAs, although after initial R&D and prototyping, ASICs are cheaper on a per-unit basis than FPGAs.

2.4. Dynamic Partial Reconfiguration of FPGAs

In an environment such as a Cloud Computing cluster which is the main focus of this work, functionality of FPGA accelerators needs to change depending on the demands of the users or background services that process data. One solution to avoid reprogramming is to house all accelerated tasks in an FPGA and leave it running indefinitely, however this brings with it immense inefficiencies in idle power consumption and drastically increases resource requirements of the FPGA platform. As such reprograming a smaller FPGA with the modules that it needs at any one time is preferable.

One method of reprogramming the FPGA involves downloading a full bitstream, either via a standard interface such as a JTAG port or from volatile memory where it is preloaded at boot time and fetched on demand. However, full bitstream reprogramming brings with it a host of disadvantages such as

1. **FPGA logic shutdown** – all operations conducted by other services or users in the Programmable Logic must terminate when full bitstream reprogramming takes place. This can cause users to see drastic performance decrease when multiple users are requesting different acceleration tasks at once.
2. **High reconfiguration overhead** – reprogramming a full bitstream, both in terms of time and energy consumed is non-trivial and needs to be minimized where possible
3. **Increased design complexity** – a full bitstream may implement a wide range of algorithms and each algorithm may be present in the bitstream as a different PE (Processing Element). This can result in an increased design complexity in cases where we want a platform that can be extended and updated with new functionality during its lifecycle and that many different users may utilize to accelerate tasks that may not coincide. This forms a combinatorial problem where repetitions of modules are allowed (a user may request the same module another user has requested) and order doesn't matter (a user doesn't care which partition region houses the module he wants to accelerate). The formula for this case is

Equation 1 – Unordered combination equation for calculating the total number of possible bitstreams that can serve any combination of tasks to carry out

$$\frac{(r+n-1)!}{\mathrm{r!\,(n-1)!}}$$

Where

a. r is the maximum number of PEs that can fit in the FPGA
b. n is the total number of tasks that have been developed to be accelerated

If we imagine a scenario where we have developed 8 different algorithms(n = 8) to be integrated in an FPGA hardware design and all 8 PEs require more or less the same number of resources. If we use a FPGA chip that can house 8 of these modules (r = 8), the number of different full bitstreams that must be generated to be able to delegate any combination of requested tasks is 6435. If we take into account that typical full bitstreams range from 2 MB to several tens of MBs, the memory requirements to allow such a system to store all these bitstreams would be in the several hundreds of GBs. In an environment where different users use the FPGA platform as an acceleration service and where each user may request a different task to be delegated, this can prove infeasible. Alternatively, and much more realistically, an FPGA can simply implement a single bitstream where each algorithm is expressed once as a single Processing Element; however multiple acceleration calls to this coprocessor from different users would cause congestion and performance decrease.

4. **High device area requirements** – even if we employ a large FPGA chip to avoid the pitfalls of the previous points by housing all modules multiple times to avoid reconfiguration, this still leads to the implementation of an inefficient, power-demanding platform that is mostly underutilized.
5. **Difficult to maintain and update** – updating a bitstream with new functionality or correcting a mistake made at design time is much harder for full bitstreams than for partial bitstreams. If a small part of the full bitstream needed change, the whole design has to be redesigned and updated. In the case of partial bitstreams, most of the time only the IP Block with the erroneous behavior needs to be changed and redeployed.

Dynamic Partial Reconfiguration is a development paradigm on the field of FPGA design whereupon specific parts of the Programmable Logic are marked as dynamically reprogrammable. This means that while an FPGA is executing tasks, a partial section of the FPGA can be hot-swapped with other generated partial bitstreams, essentially reconfiguring that subsection of the FPGA with a different functionality.

Benefits of DPR over static programming of an FPGA are immediately apparent

1. **FPGA logic uninterrupted during reconfiguration** – all operations conducted by other services or users in the FPGA can continue uninterrupted while a specific Partial Region is being reconfigured. This is important for mission-critical applications or to decrease the chance a user will experience slowdown of requested services. This capability is true for the case of Dynamic PR only. Static Partial Reconfiguration (SPR) is the similar to DPR, with the exception that the device must shutdown for the duration of the reprogramming stage.
2. **Low reconfiguration overhead** – reprogramming a partial bitstream is significantly faster and incurs much smaller energy consumption and time overhead than programming a full bitstream. This is especially apparent for large FPGA devices where a programmable region may be a small percentage of the overall full bitstream. Of course, this overhead is still non-zero and needs to be taken into account, and in this sense intelligent scheduling and resource reuse is important to reduce the chance partial reconfiguration is needed.
3. **Significantly reduced design complexity** – a DPR-enabled hardware design requires only that each partition marked as reconfigurable implements the reconfigurable module intended to run in it at runtime. In the case of Cloud Computing environments we assume in this work that all regions should be able to house any accelerated task. In Xilinx's DPR methodology, each partition requires its own copy of a partial bitstream for a given task implemented. A generated full bitstream that has DPR enabled can house at any time, any partial bitstream that was generated with this design as reference. As such only a single full bitstream needs to be loaded at boot time and any functionality can be loaded on demand later from the partial bitstreams. This means that for 8 regions marked as reconfigurable (r = 8) that can house any of 8 accelerated tasks (n = 8) the total number of partial bitstreams needed is 64. The general equation describing the memory requirements of storing all partial bitstreams of a DPR FPGA design is the following

Equation 2 – Equation for calculating the size of all partial bitstreams in a Dynamic Partial Reconfiguration design where all Reconfigurable Regions can house any of the implemented algorithms

$$PartialSize_{tot} = \mathrm{n} * \sum_{i=1}^{r} PartialSize_i$$

Where

$PartialSize_{tot}$ is the total size of all the partial bitstreams generated by the design measured in bytes

n is the number of accelerated tasks developed

r is the number of Reconfigurable Partitions defined at design time

$PartialSize_i$ is the size of the partial bitstreams that program Reconfigurable Partition i measured in bytes

Assuming near full utilization of the FPGA's resource from the 8 regions, the calculated total memory needed for the 64 partial bitstreams is close to $n*S_{full}$ where n is the number of implemented modules and S_{full} is the size of the full bitstream.

4. **Easier to maintain and update functionality** – updating functionality of deployed partial bitstreams and correcting errors made at design time is much easier for DPR-enabled platforms. The caveat to this is that the newly developed partial bitstreams must be able to fit into the region marked by the full bitstream; else a new redesign process needs to be done where Reconfigurable Partitions are resized or the whole platform is migrated onto a bigger FPGA chip. In order to avoid this, overprovisioning of FPGA resources in Partial Reconfiguration Regions can be employed.

Benefits of employing dynamic partial reconfiguration include

1. The ability to time-multiplex tasks on an FPGA by swapping functions in and out, reducing area and power requirements
2. Allow flexibility in algorithms and subtasks available to applications
3. Allowing uninterruptible workload execution of static logic, useful for multi-tenant systems
4. Accelerated Reconfigurable Computing
5. Updating hardware functions marked as reconfigurable can be done easily and remotely.

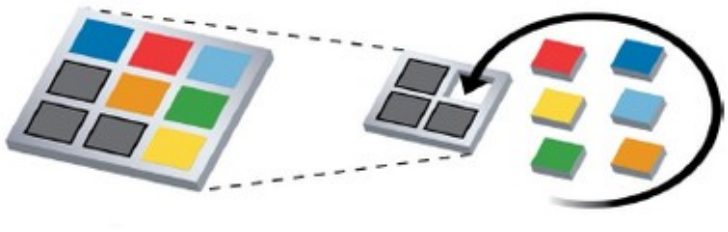


Figure 2 – Time multiplexing of multiple functions on a single reconfigurable partition reduces area requirements

2.5. Energy Efficient Computing

In this work, power consumption of the proposed system is an important metric that needs to be measured. This is necessary in order to accurately evaluate the benefits that FPGA-based cloud computing can offer to businesses and organizations.

Information and Computer Technology (ICT) offers the capability to increase efficiency of resources utilized in industrial and enterprise environments by optimizing and automating processes [16]. ICT infrastructure itself however needs

energy to operate and when employing such solutions it is important to apply the same principles of energy efficiency on the infrastructure itself.

Development of energy efficient computing solutions can help companies save money by reducing electricity bills, permit battery-powered devices to operate for longer times before needing recharge (such as for mobile phones, laptops and sensor devices) or before they are decommissioned from service (for one-time deployed devices such as IoT devices).

In the case of Cloud Computing infrastructure, these energy savings can help reduce costs of maintaining such infrastructure as well as meet criteria of legislation on eco-friendly design of ICT infrastructure such as (EU) No 617/2013 implementing Directive 2009/125/EC [17].

Research work on the impact of ICT on global energy consumption has been conducted extensively in the last years.

In 2015, Andrae and Edler [18] published their work on energy consumption trends in the ICT sector. In their work, they presented categorized estimations of worst-case, expected and best-case scenarios of power consumption in TWh that various groups of ICT infrastructure would incur. If left unchecked, GreenHouse Gases (GHG) emissions of ICT infrastructure could contribute up to 23% of global GHG emissions. Reducing energy usage can help reduce GHG emissions that are incurred as a byproduct of the generation of electrical power that feeds these devices and data centers.

There are 2 main procedures involved in implementing energy efficient computing platforms.

1. Power consumption measurement of the device at runtime, both at full capacity usage as well as while in idle mode
2. Redesigning the platform both at system level as well as at the application level to decrease power consumption of the system

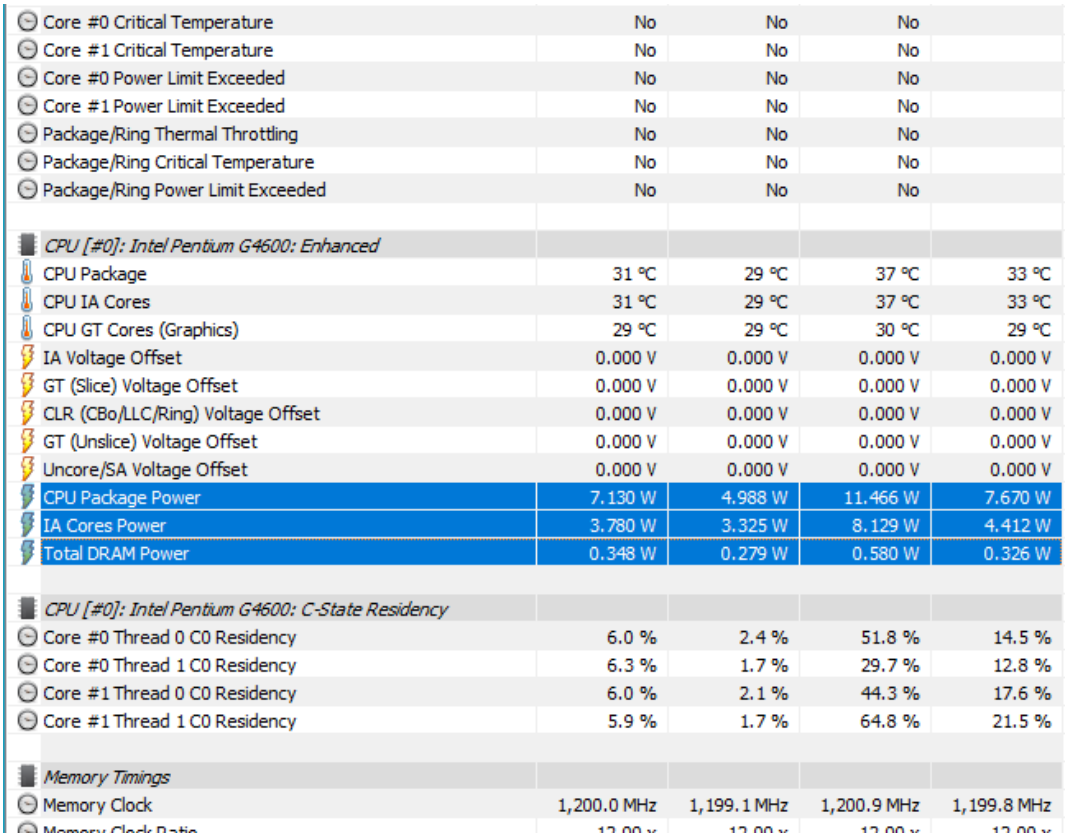
There are several methods described in the literature that are used to measure power consumption in computing systems.

In 2019, Fahad et al. [19] published a comparative study of methods that can be utilized to measure power consumption of computing systems.

One method is using external power metering devices that measure the power consumption of a workstation as it operates. This method offers relatively accurate results but is only suited for system-wide power sensing, not allowing for highly granular, component level power consumption measuring, such as measuring the energy consumption of the CPU or the GPU.

An example of an external power metering device is the Kill-a-Watt [20]. Kill-a-Watt is an electricity usage monitoring device manufactured by Prodigit that acts as an intermediate between devices and wall-mounted power sockets and measures power that a connected device draws.

A second method used for measuring energy consumption is using power sensors installed on CPUs and GPUs at manufacturing time. These sensors provide a more granular power measuring methodology than using external power meters.



Core #0 Critical Temperature	No	No	No	
Core #1 Critical Temperature	No	No	No	
Core #0 Power Limit Exceeded	No	No	No	
Core #1 Power Limit Exceeded	No	No	No	
Package/Ring Thermal Throttling	No	No	No	
Package/Ring Critical Temperature	No	No	No	
Package/Ring Power Limit Exceeded	No	No	No	
CPU [#0]: Intel Pentium G4600: Enhanced				
CPU Package	31 °C	29 °C	37 °C	33 °C
CPU IA Cores	31 °C	29 °C	37 °C	33 °C
CPU GT Cores (Graphics)	29 °C	29 °C	30 °C	29 °C
IA Voltage Offset	0.000 V	0.000 V	0.000 V	0.000 V
GT (Slice) Voltage Offset	0.000 V	0.000 V	0.000 V	0.000 V
CLR (CBo/LLC/Ring) Voltage Offset	0.000 V	0.000 V	0.000 V	0.000 V
GT (Unslice) Voltage Offset	0.000 V	0.000 V	0.000 V	0.000 V
Uncore/SA Voltage Offset	0.000 V	0.000 V	0.000 V	0.000 V
CPU Package Power	7.130 W	4.988 W	11.466 W	7.670 W
IA Cores Power	3.780 W	3.325 W	8.129 W	4.412 W
Total DRAM Power	0.348 W	0.279 W	0.580 W	0.326 W
CPU [#0]: Intel Pentium G4600: C-State Residency				
Core #0 Thread 0 C0 Residency	6.0 %	2.4 %	51.8 %	14.5 %
Core #0 Thread 1 C0 Residency	6.3 %	1.7 %	29.7 %	12.8 %
Core #1 Thread 0 C0 Residency	6.0 %	2.1 %	44.3 %	17.6 %
Core #1 Thread 1 C0 Residency	5.9 %	1.7 %	64.8 %	21.5 %
Memory Timings				
Memory Clock	1,200.0 MHz	1,199.1 MHz	1,200.9 MHz	1,199.8 MHz

Figure 3 - Example of software monitoring system operational metrics such as wattage and temperature.

Finally, a third method is based on software that utilizes energy predictive models that take as input operational metrics such as FLOPS and cache miss rate and calculate an estimated energy consumption. These energy prediction models have been researched thoroughly, however they have been found to be inaccurate often due to the fact that there are a large number of available models to choose from and selecting the subset most relevant to interpreting power rating is difficult. These models can sometimes be inaccurate and/or incomplete in describing the power requirements of the device they attempt to model.

One example of software implemented energy consumption forecasting is Xilinx Power Estimator (XPE) [21] and Vivado Power Report tools.

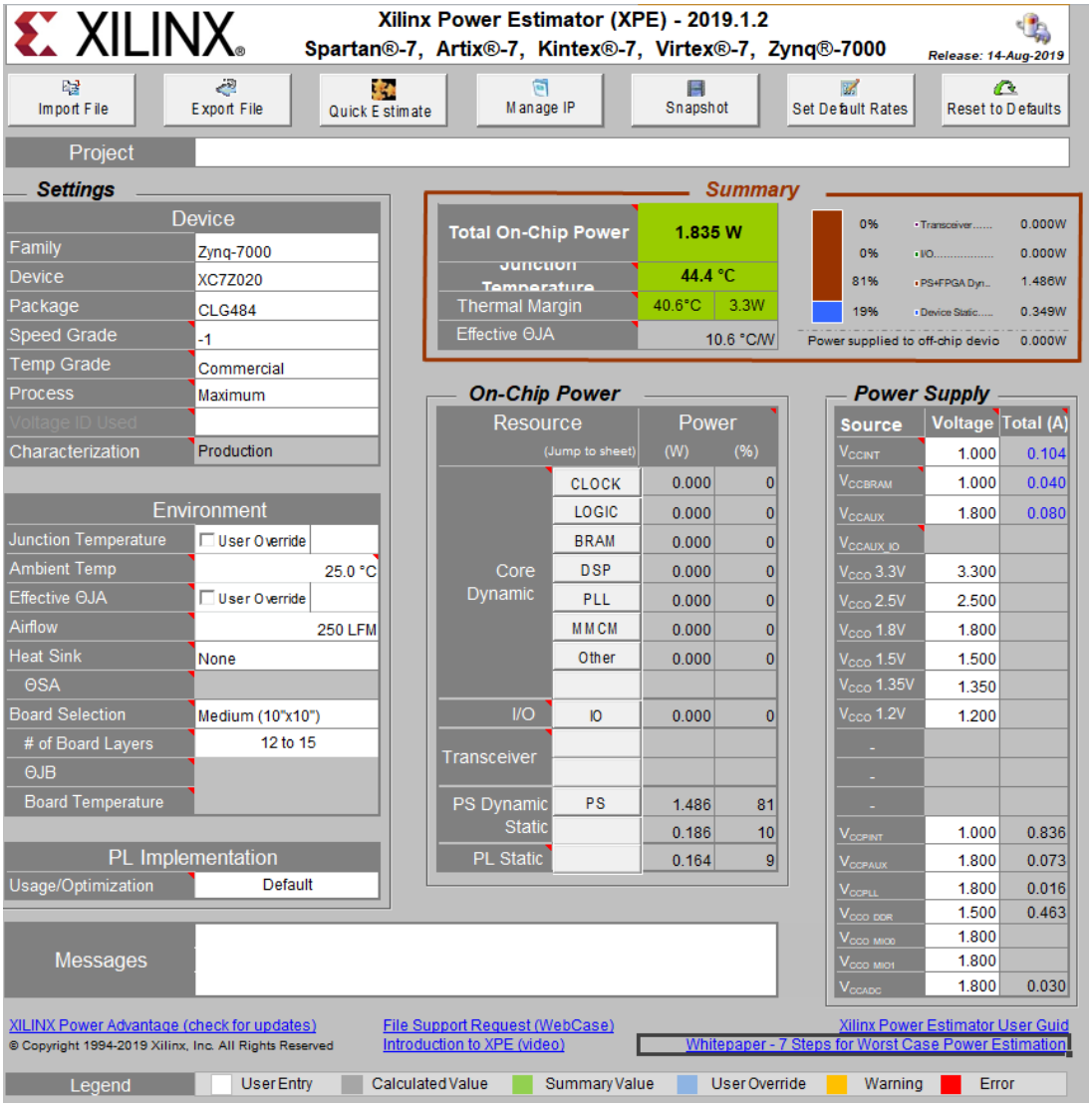


Figure 4 – Summary overview of a Zynq-7000 device with partially user-defined workload parameters

XPE is a set of Excel-based spreadsheets created by Xilinx which is used to calculate power consumption of Xilinx FPGA devices in a pre-synthesis design stage. The developer can input the model of the device he wishes, workload of independent FPGA and peripheral devices in a highly granular manner, read and write rates of off-chip DDR memory and other parameters. The tool will then estimate the expected wattage of the device and give component-level power consumption metrics as well.

Design techniques utilized for conserving energy have been employed that are unique to FPGA environments. As noted before, Dynamic Partial Reconfiguration is a system level energy optimization method that allows a decrease of required energy resources by reducing device area and subsequently power requirements as well as allowing the programming of blank partial bitstreams that further reduces energy consumption in an idle platform. This alongside an implementation of a DPR-aware scheduler to minimize the probability that reconfigurations take place can further help in meeting both performance and energy efficiency goals.

2.6. Related Work on FPGA-based Cloud Computing

In recent years, the viability of utilizing FPGAs in Cloud Computing environments has been recognized and researched upon. Cloud computing solutions are increasingly utilized to address processing needs in big data and data acquired from web services, multicore computing systems [22] and intelligent IoT Devices [23].

Moreover, various techniques have been the focus of research for on-chip and system monitoring for power and energy efficiency [24], [25] and additionally to manage SoC power and energy [26], [27], [28]

In 2011, Yu et al. [29] proposed a web server implementation that utilizes the FPGA of the BEE3 multi-FPGA chassis system to carry out web data processing using a Microblaze Soft-core processor [12] and a custom Web Processing Module that handles tasks such as TCP packet decomposition and URL parsing. The proposed implementation offered up to 4 times higher performance per watt while maintaining an overall comparable performance to a Xeon 5520 implementation of the web server.

In 2012, Eguro and Venkatesan [30] present a system architecture based on an FPGA for trusted cloud computing applications that emulates homomorphic encryption [31] by providing a safe area in the FPGA that allows secure processing of sensitive data.

In 2015, Fahmy et al. [32] present a model of a platform for integrating virtualized accelerator modules of FPGAs to existing cloud computing infrastructure in order to ensure high efficiency and performance goals. In their proposed model, they use Partial Reconfiguration and a scheduler to dynamically reprogram partitions in the PL according to the user requests and maximize usage of FPGA resources.

In 2018, Vaishnav et al. [33] introduced the concept of resource elasticity by enabling the reallocation of FPGA spatial resources using OpenCL and dynamic partial reconfiguration to allow higher performance and resource utilization of FPGA accelerator platforms. Their experiments on different types of scheduling schemes for allocating accelerator resources showed that cooperative scheduling is a better method for FPGA platforms.

FPGA based cloud computing solutions often referred to as FPGA as a Service (FaaS) have been proposed in the literature in recent years and early commercial implementations have shown promising results, both for businesses as well as for end users.

With their EC2 F1 cloud service, AWS [34] aims to provide a flexible computing environment of Virtex Ultrascale+ family FPGAs alongside a development environment. Amazon's EC2 is characterized by the capability of designing and deploying a variable amount of FPGA platforms running hardware designs created by developers on Xilinx's IDEs integrated in the Amazon ecosystem [35].

Alibaba has also released commercially viable cloud computing resources utilizing FPGAs as computing platforms that cloud users can employ for their needs [36]. The FaaS provided by Alibaba features 2 different instances of F1 and F2 instances providing both Intel and Xilinx small-scale devices for customers with ease of deployment.

2.7. Related Work on Dynamic Partial Reconfiguration

Before developing a system capable of DPR in a cloud computer environment, it is important to research upon the benefits of Dynamic Partial Reconfiguration as advertised by Xilinx as well as from published research work that attempts to evaluate these benefits.

To improve efficiency of reconfigurable resources, solutions have been proposed [37]–[40].

In UG909 on Dynamic Partial Reconfiguration [41], Xilinx gives an introductory overview of DPR and what benefits it can bring to the table for FPGA developers. In this guide basic terminology and design considerations as well as some example applications which could benefit from DPR are presented.

Nguyen et al. [42] present their findings in evaluating and quantifying the benefits that DPR can offer to embedded vision applications when compared to static FPGA design methodologies. Power savings of up to 30% can be reached by implementing DPR on a platform. Their findings show that embedded solutions that benefit from the effects of DPR share 2 main characteristics. First, all implemented tasks of the system are not needed at all times. Only 1 or a small subset of the implemented tasks needs to run concurrently at any one time. Second, the embedded solution has energy efficiency needs that need to be maximized due to the fact the device operates on batteries and area/device costs need to be minimized.

These findings coincide with the needs and nature of Cloud Computing environments. Not all implemented tasks need to be executed at all times.

Sometimes users make use of other resources that do not require FPGA resources such as File I/O or environment settings management. A given user's requested tasks have no effect on when and how often other users request tasks.

Secondly, DPR can increase energy efficiency of the cloud computing cluster by implementing the design in a smaller FPGA chip that can time-multiplex the requested tasks in the partial regions defined. In addition, reprogramming a blank

bitstream inside a region not utilized can further decrease power consumption when in idle mode.

Nafkha and Louet [43] researched upon the overhead of power consumption when DPR is employed in a platform. In their work, 94 KB sized partial bitstreams are programmed through the ICAP interface at runtime, increasing the power consumption from 340 mW to 500 mW for the duration of the reconfiguration. In our work, partial bitstreams are larger (700-1100 KBytes) and the reconfiguration overhead is much bigger when compared to the execution time of the typical size of data users may request.

This brings up an important metric that needs to be taken into account when designing a DPR-enabled FPGA platform. This metric is the Execution-to-Reconfiguration (ER) ratio of execution time over reconfiguration time.

Equation 3 – Equation for computing the ratio of the time spent executing to the time spent reconfiguring a reconfigurable partition in a Dynamic Partial Reconfiguration design

$$ER\ ratio = \frac{T_{execution}}{T_{reconfiguration}}$$

For example, if the partial bitstream requested to be reconfigured takes 10 msec to be programmed and the programmed module runs for 5 msec on requested data, the ER ratio is 0.5.

High ER ratios indicate the partial reconfiguration is a small overhead in the computing process. Low ER ratios indicate a high reconfiguration overhead. This means that reconfiguration needs to happen sparsely in order to meet energy efficiency and performance goals.

Luo et al. [44] proposed a DPR model of FPGA platform that utilizes the multi-threaded nature of a Linux operating system to delegate tasks to HW accelerators at runtime in an efficient, demand-driven manner. The aim is to solve the lack of Partial Reconfiguration enabled platforms operating under Linux. Linux-based DPR development could decrease the time needed to create an efficient FPGA acceleration platform and increase design flexibility thanks to the widespread support and contributions of the Linux community.

3. Proposed System Architecture and Development Environment

In this chapter, the system architecture is presented for the developed platform as well as information on the development environment and specifications of the FPGA platform used to develop the system.

3.1. Proposed System Architecture and Flow

Below is an architectural diagram of the implemented system on the Zedboard SoC.

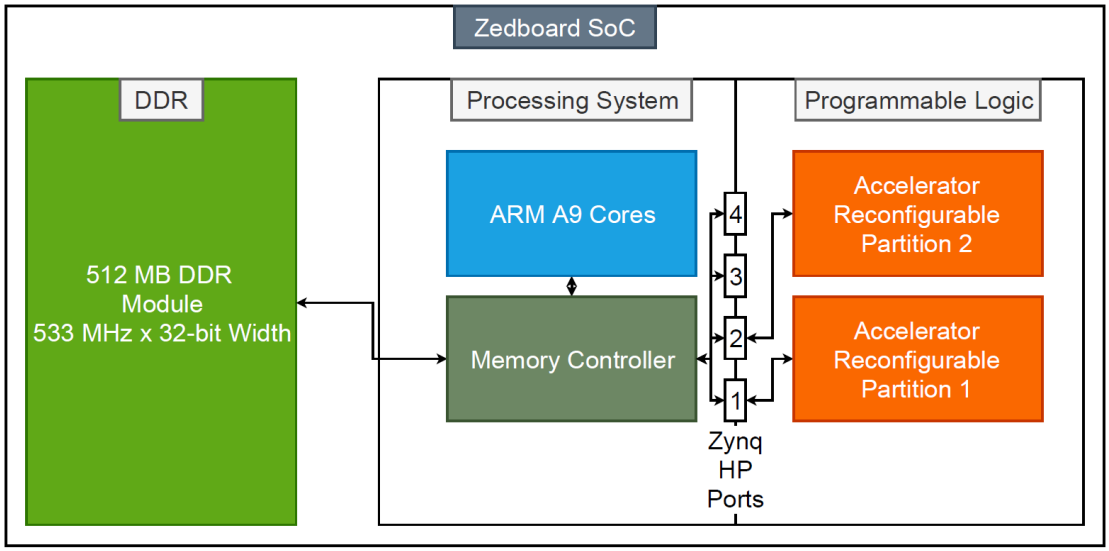


Figure 5 – Proposed System Architecture Diagram

In the proposed system shown in Figure 5, the Reconfigurable Partitions (RPs) 1 and 2 are regions of the FPGA fabric that are defined post-synthesis to house any functionality inside so long as it covers the following criteria

1. The function implemented in the partition, named a Reconfigurable Module (RM) must utilize fewer logic resources than the resources allocated to the reconfigurable partition at design time.
2. The RM added to the platform on a RP (Reconfigurable Partition) must have the exact same interfacing as the interfacing of all other RMs that can be programmed on the RP. For the purposes of our system, all RPs are capable of housing any of our implemented algorithms on the FPGA. This means that both RPs have the same interfacing logic.

Each reconfigurable partition can house any of the developed HLS IPs inside. Implementing and generating bitstreams for a design where DPR has been enabled leads to the generation of partial bitstreams. These partial bitstreams are the files that need to be programmed to the PL at runtime in order to program the capability of each function in the fabric.

Because the partial bitstreams implement only part of the FPGA, they are much smaller than the full bitstream. The size of a partial bitstream is directly proportional to the size of the PL that partition is allocated on at floorplanning time.

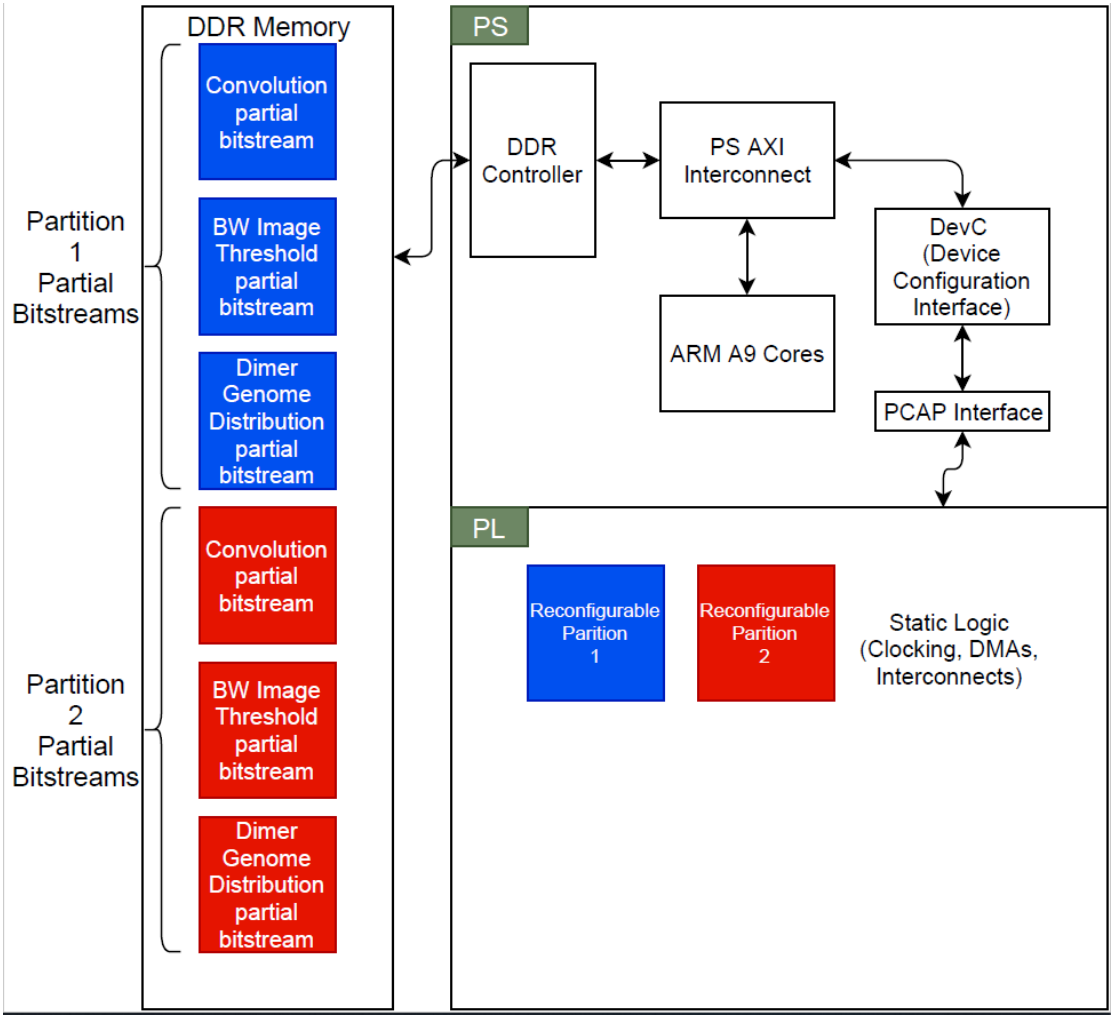


Figure 6 – Operational snapshot of the proposed system.

Generated partial bitstreams are loaded from the SD card on the DDR at system boot time. In Figure 6, Partition 1 partial bitstreams are only compatible with partition region 1 (blue). Different partial bitstreams need to be generated for partition 2 (red) even if they implement the same functionality.

3.2. FPGA Platform System Specifications

The proposed system is developed and evaluated on a Zedboard All-Programmable SoC development platform [45]. The Zedboard is a development board featuring both an ARMv7 CPU and an FPGA chip. It is equipped with a 512 MB DDR3 memory module clocked at 533 MHz and an interface width of 32 bits. The CPU is a dual-core ARM A9 and the FPGA chip is a Zynq-7000 family chip, the XC7Z020-CLG484.

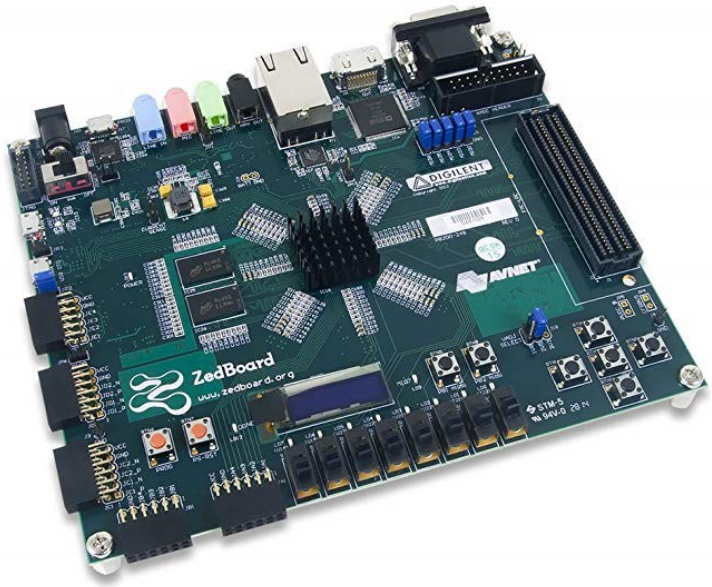


Figure 7 – The Zedboard, choice of implementation for proof-of-concept

The ARM A9 Dual-Core CPU is a low power processing unit used mainly in embedded circuits. Its TDP is rated at approximately 0.25 W per core..

The FPGA chip of the Zedboard is comprised of a moderate amount of programmable logic that is mostly suited for evaluating applications at a small scale before porting them to larger, more resourceful FPGA chips. The table below shows the number of Flip Flops, Look-Up Tables, Digital Signal Processors and Block RAM available in 36Kbit tiles and in KBytes that the XC7Z020-CLG484-1 is comprised of.

Table 1 – Target FPGA platform available resources

FPGA Model Name	LUTs	FFs	DSPs	BRAM36 Tiles	BRAM (in Kbytes)
XC7Z020-CLG484-1	53200	106400	220	140	615

Besides resource availability, architectural specifications of the Zynq-7000 need to be taken into account when designing an accelerator IP. The Zynq-7000 family of SoCs features a PS-PL interface that allows FPGA IPs to access the DDR at a configurable rate. More specifically, the interface options between the Programmable Logic and the Processing System is comprised of the following

- Accelerator Coherency port (ACP) (1 port, 64-bit width, cache coherent memory accesses)
- High performance (HP) PL interfaces, (4 ports, 32 or 64 bit width, non-cache-coherent accesses)
- General purpose PL interfaces (GP) (2 ports, 32-bit width, no FIFOs meaning lower performance than HP ports)
- Device configuration (DevC port, used for configuring the device at runtime)

In this work, 2 HP ports and a single GP port are used to connect to reconfigurable partitions (1 port per partition) to the PS side and 1 GP port is used, connected to both partitions.

Each HP port has a dedicated channel for receiving and transmitting data. The HP ports are responsible for transferring the main bulk of the data to be processed at the PL as well as the algorithm-specific metadata for each application (e.g. the kernel values of the convolution filter). The GP port is used to transfer only the size of the input data and the output data and is used to transfer this data on both partitions.

Besides the width of the port used to transfer data, the rate at which the data is transferred is also important. In this case, the PL clock is set at 7ns period or at a frequency of approximately ~143 MHz. This coupled with the width of 8 bytes of the HP port creates a maximum theoretical bandwidth of 1089.9 Mbytes/sec. This is calculated from the equation

Equation 4 – Maximum bandwidth of PS-PL ports on the Zynq-7000 family of devices

$$Max\ BW = f * w$$

Where:

BW is the maximum theoretical bandwidth (measured in bytes/sec)

f is the bus frequency (measured in Hz)

w is the width of the bus (measured in bytes)

This upper limit is a very important metric that is used to verify the correct setup and operation of the FPGA accelerator platform and evaluate the viability of migrating a task to the FPGA in early design stages.

3.3. System Design and Development Environment

The development of the proposed system was carried out on a Linux CentOS 7 workstation running on an Intel i5. The software tools utilized for designing the hardware platform on the Zedboard are Xilinx Vivado HLx 2017.4 Suite. Specifically, 3 different IDEs were utilized

1. Vivado HLS 2017.4 for implementing the 3 algorithms in C++ and compiling to RTL code
2. Vivado 2017.4 for designing the DPR platform with the 3 algorithms designed on Vivado HLS and enabling their dynamic reconfiguration on 2 separate reconfigurable partitions.
3. Xilinx SDK 2017.4 for developing the baremetal application responsible for system initialization, scheduling of requested tasks to be programmed on the RPs of the FPGA, file I/O and delegating workloads on the accelerators programmed on the FPGA.

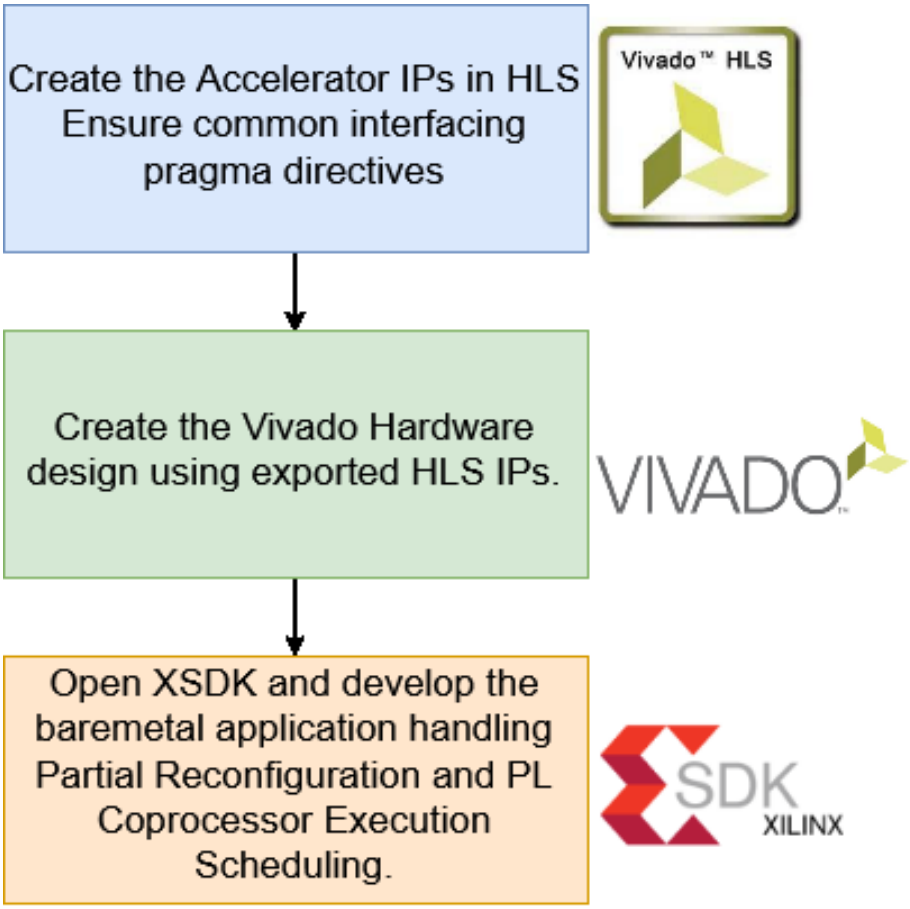


Figure 8 – Basic workflow for designing the DPR-enabled platform on Xilinx Vivado Design Suite

The reason for using version 2017.4 of Vivado was because after this version, Xilinx deprecated the driver responsible for the runtime reconfiguration of full/partial bitstreams in favor of the FPGA Manager Linux API of device-agnostic and manufacturer-agnostic bitstream programming of FPGA devices running on a Linux OS [46].

However, during development of the platform on later versions of Vivado, issues arose due to Xilinx not having released partial bitstream reprogramming functionality of these drivers for the Zynq-7000 family of chips. As such, the latest version of Vivado Design Suite that supported the xdevcfg driver was used.

In Vivado HLS, the applications were developed using the C++ programming language. IDE-specific libraries that are designed by Xilinx such as HLS_Stream and Arbitrary-Precision Integers were utilized [47] [48]. These libraries contain functions and data objects optimized for implementation in an FPGA and can help developers reach QoR goals.

4. Implementation Methodology

4.1. Multidisciplinary Algorithms Implemented

In this chapter we present the 3 algorithms that were developed to run on the Zedboard FPGA platform. These algorithms are

1. A black and white thresholding (BW Threshold) algorithm for grayscale images. The user can select the threshold to use. The threshold value is in the range [0,255].
2. A 3x3 image convolution filter for grayscale images. The user can select the type of kernel to use in the convolution filter.
3. A dimer global base distribution algorithm that measures the distribution of nucleotides of length 2 in DNA genome sequences.

In order for all 3 modules to be interchangeable in the reconfigurable regions, their interfaces were developed to be exactly the same.

```
void bw_thres(
        AXI_STREAM & input,
        AXI_STREAM & output,
        int sizeIn,
        int sizeOut) {
```
(a)

```
void convolution(
        AXI_STREAM & input,
        AXI_STREAM & output,
        int sizeIn,
        int sizeOut) {
```
(b)

```
void dimer_genome_distribution_64bit(
        AXI_STREAM & input,
        AXI_STREAM & output,
        int sizeIn,
        int sizeOut) {
```
(c)

Figure 9 – Interface of the implemented algorithms as defined in C++ source code. Note that the port names and types are exactly the same. This is important for enabling dynamic partial reconfiguration.

All 3 algorithms have the exact same interface definition to allow Dynamic Partial Reconfiguration. The AXI_STREAM type is a custom-defined structure of an AXI stream type structure exclusive to Vivado HLS with a user-defined data width of 8 bytes (64-bit width).

Both input and output streams are defined as 64-bit width ports to be used for transferring and receiving data. The sizeIn and sizeOut parameters are 32-bit integer values used to declare the number of 64-bit input and out elements respectively to be transferred. sizeIn and sizeOut variables are transferred to the HLS IP blocks using the AXILite interface through the GP port.

As mentioned before, one of the main goals of this project is the development of an acceleration platform for multi-disciplinary tasks. Since interfaces in a DPR-enabled platform's RPs must be identical among RMs, relaying each individual IP's parameters using the AXILite interface is inefficient.

In light of this, the selected algorithms were developed in Vivado HLS to use the input AXI_STREAM port to receive the parameters (such as the 3x3 kernel for the convolution kernel) just before receiving the main data input from the A9 CPU at execution time.

In order for the transfer of the parameters to the HLS IP Block to be successful, the parameters must be passed to the IP in the exact same order they are parsed in the HLS IP Block.

For example, in the Image Convolution Filter application, we need to pass the source image's width as well as the 9 values of the 3x3 kernel. As such, the HLS IP first reads a 64-bit value from the stream and casts it as a 32-bit unsigned value to store the source image's width in local memory. Then, to read the kernel, the IP block reads 9 more 64-bit values and casts them as 16-bit signed integer values in local memory.

When the ARM A9 CPU sends this data using the AXI DMAs, the IP block assumes that they are sent in this exact order. It is the developer's duty to ensure the application running on the CPU that handles service offloading to the PL sends the parameters in the correct order. These considerations do not impact the hardware platform design flow in Vivado.

4.1.1. Image Black and White Thresholding

Black and White thresholding is an algorithm that transforms images to a format of black and white only pixels. It is a method used to partition an image to foreground and background constituents and is mainly used in object identification tasks.

The input image is transformed to a black and white image where each pixel is white if its input value is greater than a selected threshold value or black if it's smaller or equal to a selected threshold. In general

$$P_o(T) = \begin{cases} 255, & P_i > T \\ 0, & P_i \leq T \end{cases} \quad (2)$$

Where:

P_o is the output pixel value

T is the threshold value ($T \in [0,255]$)

P_i is the input pixel value

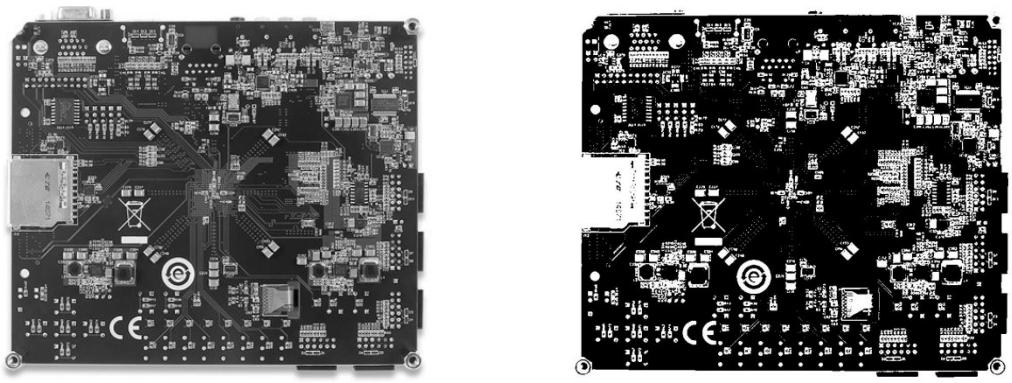


Figure 10 – Result of Black and White thresholding with threshold T = 120 applied on an image.

4.1.2. Image Convolution

The second application developed is an image convolution filter. Image convolution is the process of applying a kernel of n*n values to each pixel of an image with the goal of extracting specific features from the image. The process involves performing an element-wise multiplication of the kernel with an n*n subsection of the image where the center pixel of the sub-array is the pixel to convolve. This results in n*n products which are subsequently summed.

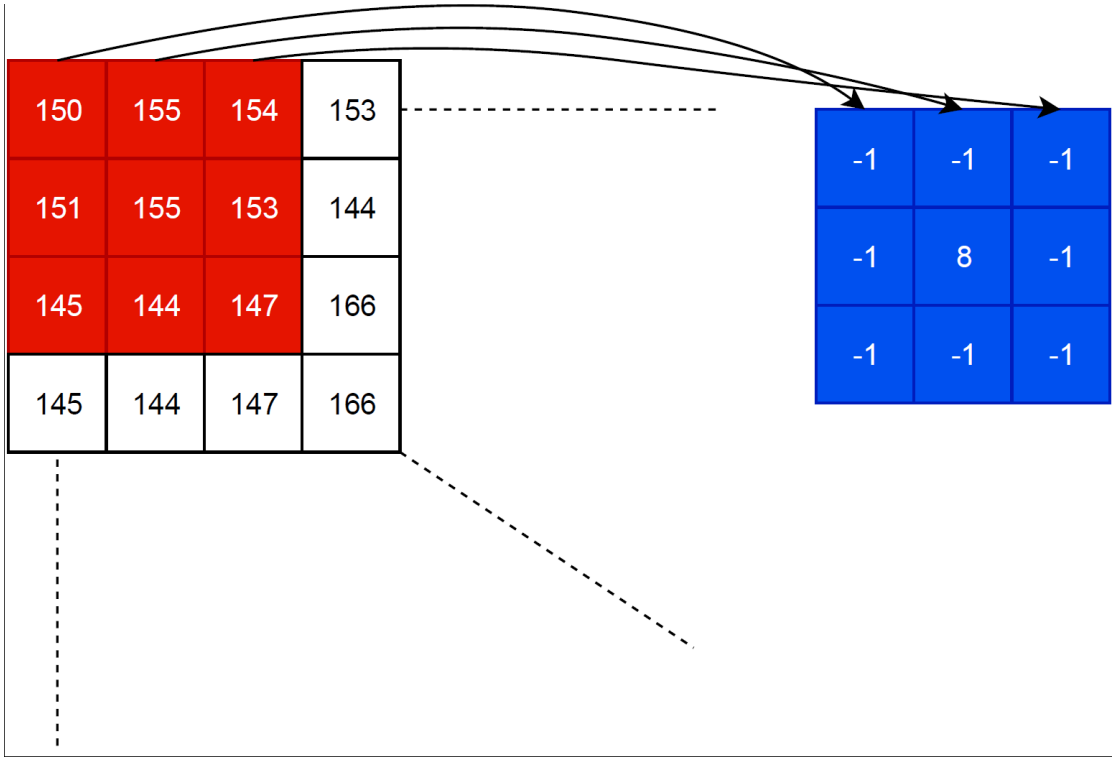


Figure 11 – Element-wise multiplication of a subsection of the source image (red) with a kernel (blue)

This sum is the convolved pixel. This number may well be a value above 255 or below 0, meaning it can't be represented correctly by an 8-bit unsigned integer for grayscale images. This problem can be resolved by simply clamping the values to the range [0,255]. Negative values are set to 0 and values greater than 255 are set to 255. This is the clamping method used in this implementation.

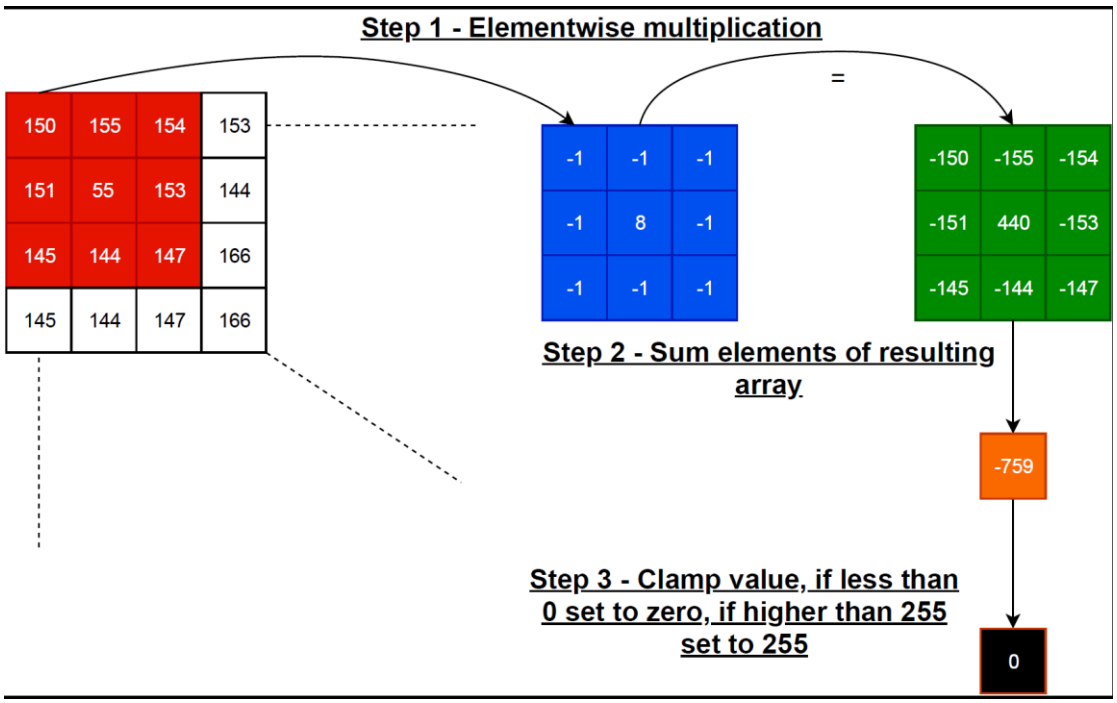


Figure 12 – Example of clamping a negative value from resulting element-wise array summation to 0

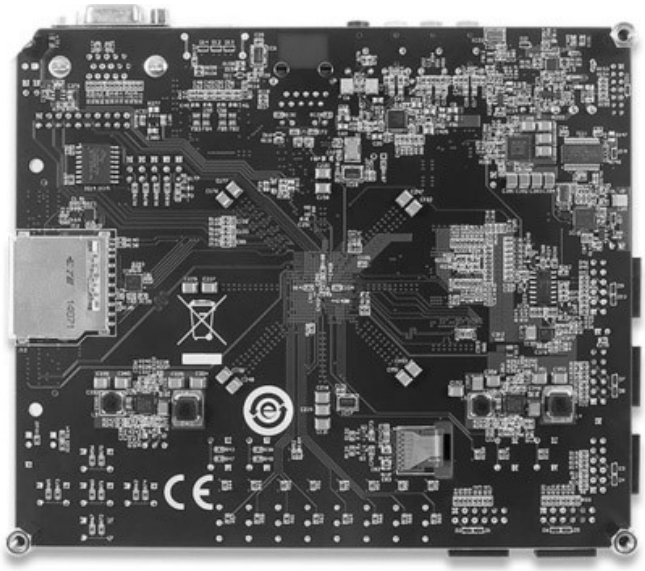
One thing to note is that depending on the size of the kernel n applied to the image, the outer region of pixels that is ((n / 2) – 1) pixels wide will not have sufficient pixels within the bounds of the image to apply the convolution on.

For example for a 3x3 kernel, the top and bottom row as well as the leftmost and rightmost column of the image will not have all necessary neighboring pixels to apply the convolution. One solution to this is to check which row and column we are currently convolving and if it's a pixel with insufficient neighbor pixels, assume 0 values for the missing pixels.

The number of rows and columns n of the kernel array should generally be an odd number. Variations for even numbers of rows/columns can be implemented but are generally avoided. The resulting image is a transformed version of the input image. The type of output image depends on the values populating the n*n kernel array.

Image convolution is usually utilized as a preprocessing task that facilitates computer vision tasks such as object identification and feature extraction, among others.

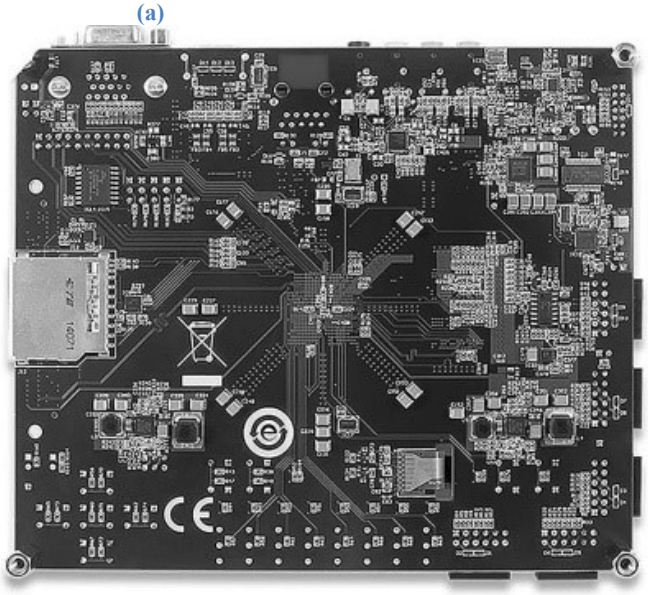
The figures below showcases the resulting images from applying 2 commonly used kernels on an image.



(a)

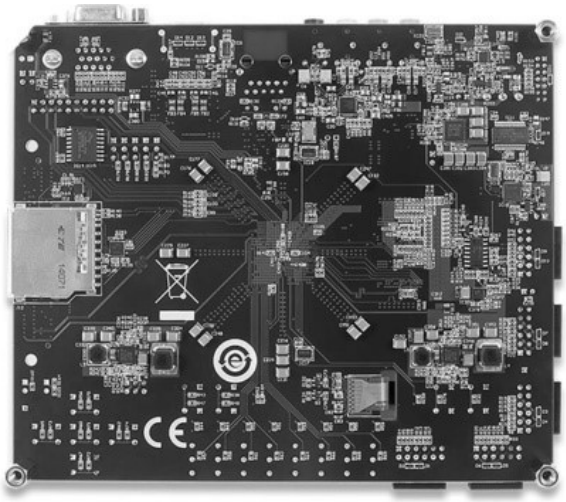
0	-1	0
-1	5	-1
0	-1	0

(b)



(c)

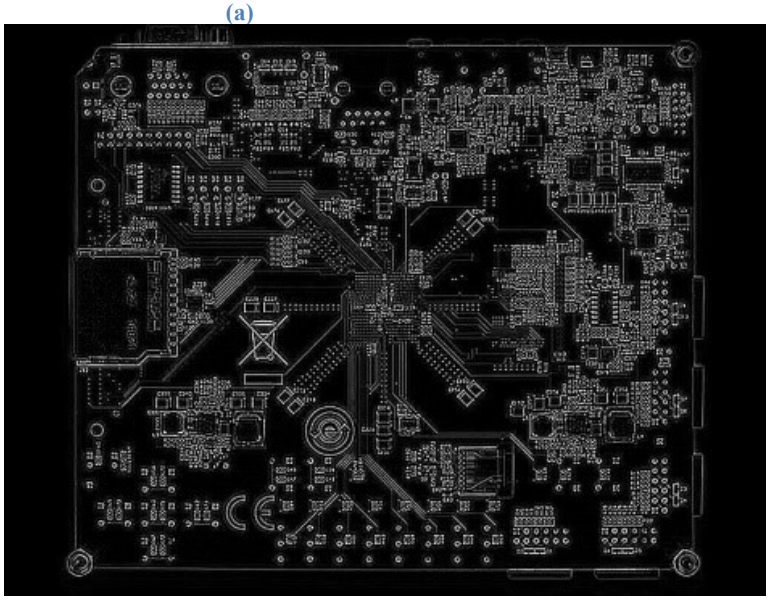
Figure 13 – Example of applying the sharpen convolution filter on an image. (a) is the input image, (b) is the kernel applied, (c) is the resulting image



(a)

-1	-1	-1
-1	8	-1
-1	-1	-1

(b)



(c)

Figure 14 - Example of applying the edge-detect convolution filter on an image. (a) is the input image, (b) is the kernel applied, (c) is the resulting image

4.1.3. Dimer Genome Distribution

Finally, the third implemented algorithm is a dimer base genome sequence global distribution counter. In the field of Computational Genomics, an n-mer base global distribution is a feature of genomic sequences that measures the relative distribution of nucleotide words of length n. Statistical analysis of the genome of an organism can offer insight to its function [49].

DNA sequences are comprised of 4 possible nucleobases, Adenine, Cytosine, Guanine and Thymine, coded for convenience as A, C, G and T respectively. This means that there are 16 possible combinations of base words of length 2.

The result of the dimer distribution is a 4x4 array where each row denotes the first base in the possible base pairs and the each column denotes the second base.

Table 2 – Dimer distribution of H. Influenza. (From [49], page 14)

	*A	*C	*G	*T
A*	0.1202	0.0505	0.0483	0.0912
C*	0.0665	0.0372	0.0396	0.0484
G*	0.0514	0.0522	0.0363	0.0499
T*	0.0721	0.0518	0.0656	0.1189

The table below shows the dimer genome distribution of the SARC-COV-2 virus. The genome sequence was taken from the RefSeq genetic sequence database [50], reference sequence NC_045512.2. The sequence was first made available from work published from Wu et al. [51] where they sequenced the virus (yet unnamed at the time of publication) from a patient working in the seafood market in Wuhan.

Table 3 – Dimer distribution of SARS-COV-2, RefSeq ID NC_045512.2

	*A	*C	*G	*T
A*	0.0964	0.0676	0.0583	0.0772
C*	0.0697	0.0297	0.0147	0.0696
G*	0.0539	0.0391	0.0365	0.0665
T*	0.0795	0.0472	0.0866	0.1075

DNA dimer base distribution is a metric that can help researchers detect unusual patterns in the genome sequence of an organism and consequently understand the structure and behavior of the organism analyzed.

For a base word of length k, we move along the genome one base at a time and check what word of length k is formed starting at each subsequent base. This means that for a genome sequence of length L, and a nucleotide word of length k, there is a total number of words W

$$W = (L - k) + 1$$

4.2. Vivado HLS Design Workflow

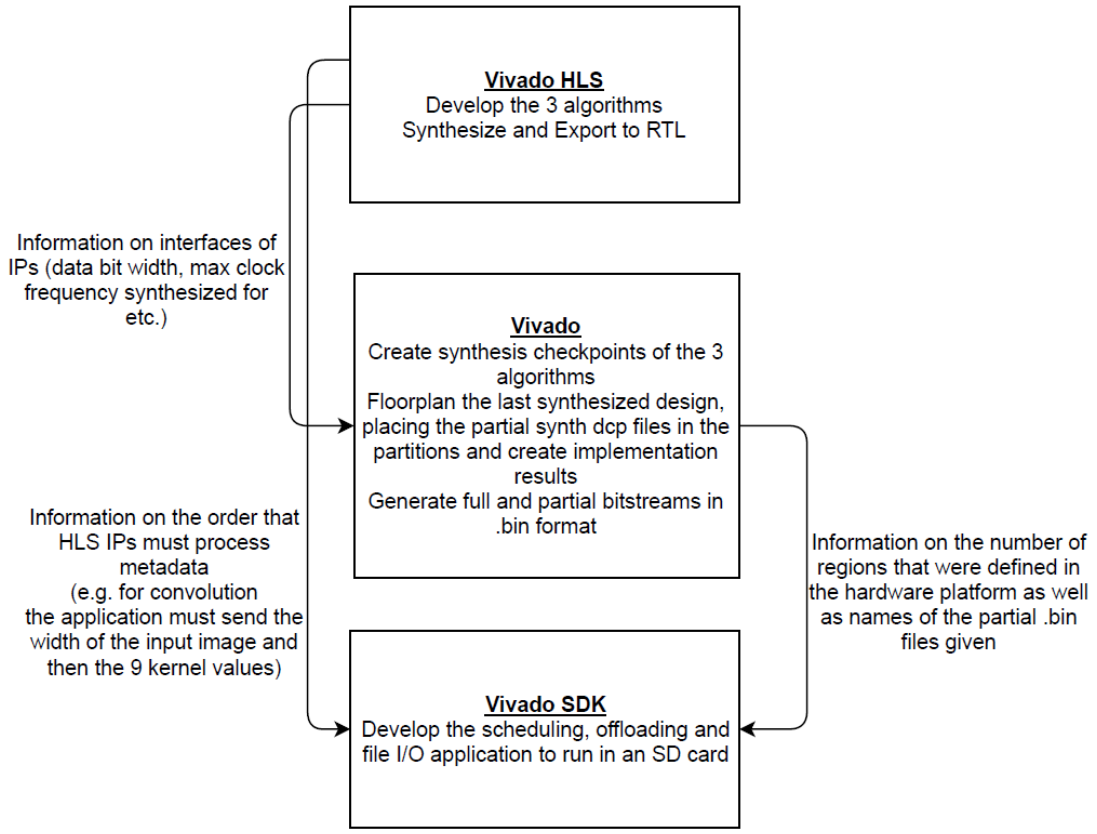


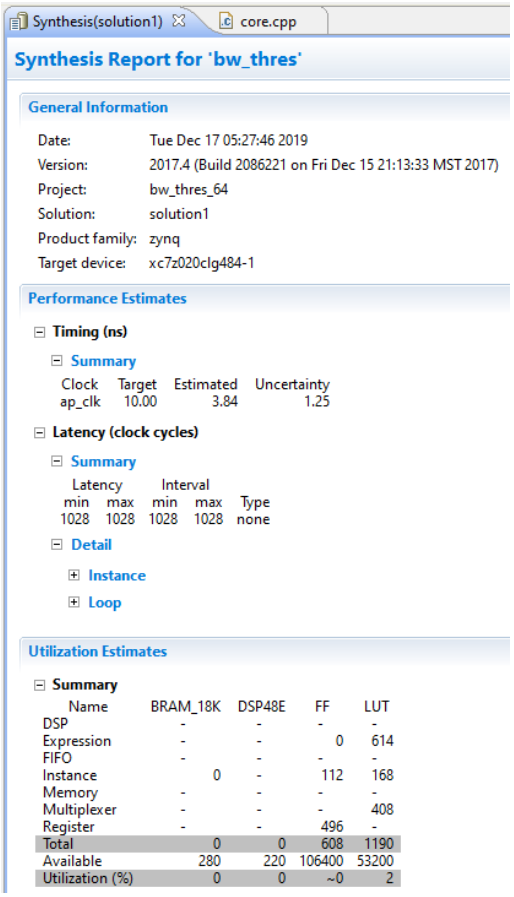
Figure 15 – Design workflow of Vivado DPR in this work

In Vivado HLS, the HLS acronym stands for High-Level Synthesis. In FPGA design, there are several methods to develop an accelerator block to be implemented in the FPGA fabric. One method is writing code to implement an intended algorithm's behavior in a Hardware Description Language (HDL) such as VHDL or Verilog. Below is an example of VHDL code for implementing a simple AND gate on an FPGA.

```vhdl
-- (this is a VHDL comment)

-- import std_logic from the IEEE library
library IEEE;
use IEEE.std_logic_1164.all;

-- this is the entity
entity name_of_entity is
   port (
         IN1 : in std_logic;
         IN2 : in std_logic;
         OUT1: out std_logic);
end entity name_of_entity;

-- here comes the architecture
architecture name_of_architecture of name_of_entity is

-- Internal signals and components would be defined here

begin

  OUT1 <= IN1 and IN2;

end architecture name_of_architecture;
```

Figure 16 – Example of VHDL code for expressing the behavior of an AND gate

Another methodology is using a higher-level programming language such as SystemC or C++ to develop the algorithm and then use a C-to-RTL synthesis tool like Vivado HLS to convert the code to HDL-equivalent format such as VHDL or Verilog and then utilize it in a FPGA Hardware Platform IDE e.g. Vivado.



Synthesis(solution1) core.cpp

Synthesis Report for 'bw_thres'

General Information

Date:	Tue Dec 17 05:27:46 2019
Version:	2017.4 (Build 2086221 on Fri Dec 15 21:13:33 MST 2017)
Project:	bw_thres_64
Solution:	solution1
Product family:	zynq
Target device:	xc7z020clg484-1

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	3.84	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1028	1028	1028	1028	none

Detail

Instance

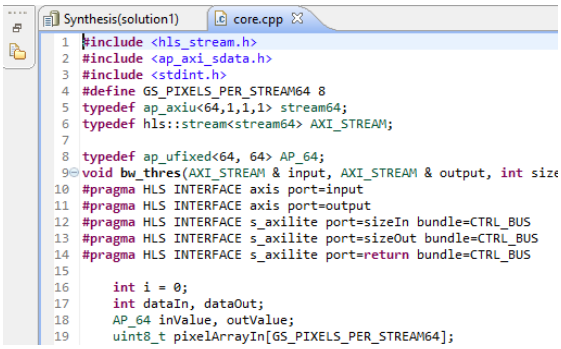
Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	614
FIFO	-	-	-	-
Instance	0	-	112	168
Memory	-	-	-	-
Multiplexer	-	-	-	408
Register	-	-	496	-
Total	0	0	608	1190
Available	280	220	106400	53200
Utilization (%)	0	0	~0	2

Figure 17 – Synthesis report for BW Threshold function in Vivado HLS. This report shows timing estimates and resource utilization estimates



Synthesis(solution1) core.cpp

```
1  #include <hls_stream.h>
2  #include <ap_axi_sdata.h>
3  #include <stdint.h>
4  #define GS_PIXELS_PER_STREAM64 8
5  typedef ap_axiu<64,1,1,1> stream64;
6  typedef hls::stream<stream64> AXI_STREAM;
7
8  typedef ap_ufixed<64, 64> AP_64;
9  void bw_thres(AXI_STREAM & input, AXI_STREAM & output, int size
10 #pragma HLS INTERFACE axis port=input
11 #pragma HLS INTERFACE axis port=output
12 #pragma HLS INTERFACE s_axilite port=sizeIn bundle=CTRL_BUS
13 #pragma HLS INTERFACE s_axilite port=sizeOut bundle=CTRL_BUS
14 #pragma HLS INTERFACE s_axilite port=return bundle=CTRL_BUS
15
16     int i = 0;
17     int dataIn, dataOut;
18     AP_64 inValue, outValue;
19     uint8_t pixelArrayIn[GS_PIXELS_PER_STREAM64];
```

Figure 18 – Sample C++ code of the BW Thresholding Function

Vivado HLS utilizes directives in the form of pragmas or tcl-based commands to allow developers to reduce latency, improve throughput or reduce resource utilization of the exported RTL code.

The most important performance enabling pragmas are

1. **#pragma pipeline –** inserted inside loop type command blocks in the C/C++ code. This directive guides the compiler to create RTL code that implements the target command block in a pipeline. If the code structure allows it, a pipeline initiation interval of 1 can be achieved, leading to immense performance increase. [52]

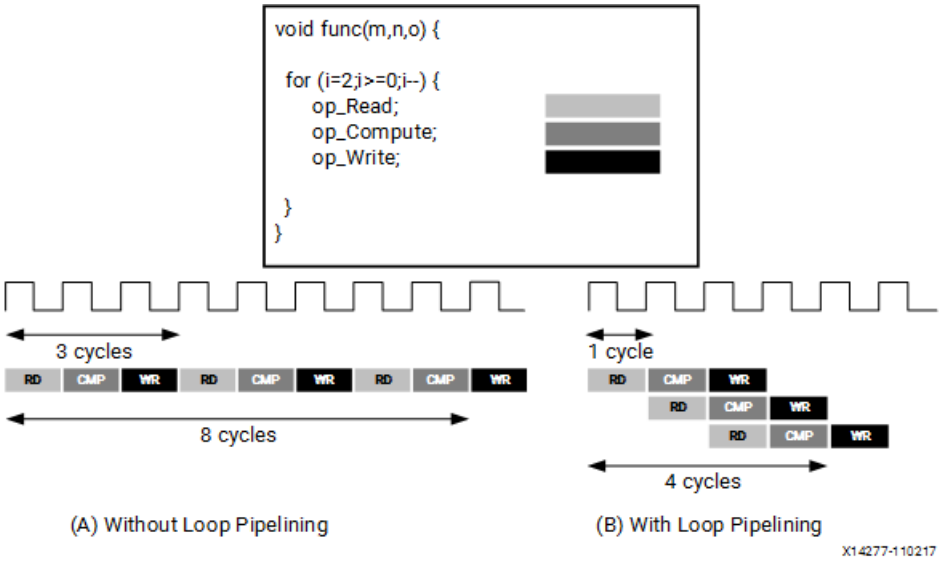


Figure 19 – Example of decreasing execution time of a loop via use of pipelining

2. **#pragma HLS partition array** – a pragma directive that is used on array variables in C/C++ code which forces the resulting Block RAM implemented RTL code to partition the array into multiple smaller arrays. This is usually applied to allow concurrent access to elements in the BRAM, either to read or to write values.

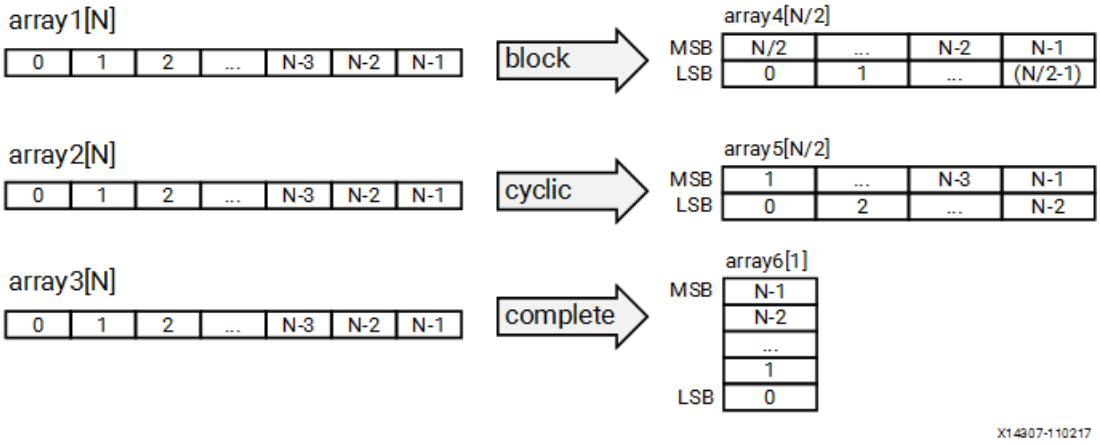


Figure 20 – Result of partitioning an array of N elements with 3 different methods.

3. **#pragma HLS unroll** – directive that is implemented in loop-type command blocks (much like the pipeline directive). This directive guides the compiler to implement RTL logic that calculates all commands in the loop concurrently.

After creating the HLS IP and synthesizing it, the next step involves exporting the IP in a format that can be imported and used in a Vivado hardware design. The user can select either VHDL or Verilog as his language of use to transform the C code to. In our case, VHDL code was selected.

Optionally, if we want to we can check the ‘Vivado synthesis, place and route’ option in the Export RTL dialog window to get a more accurate resource utilization and timing report than the HLS synthesis value, since these are estimates of the tool and may differ greatly from actually synthesized and implemented design blocks.

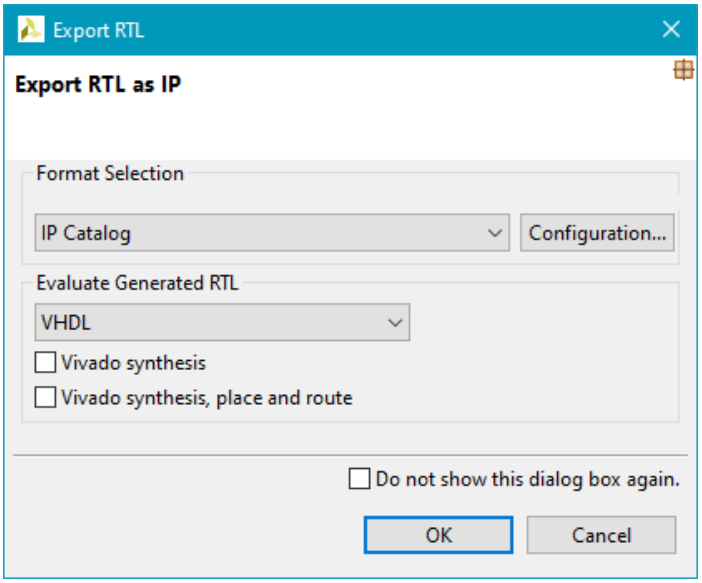


Figure 21 – Export RTL dialog box.

4.3. Vivado Design Workflow

After creating the HLS IP Blocks in Vivado, the next step is to create a hardware platform in Vivado and integrate the accelerator blocks with DMA controllers and AXI peripheral interconnects and generate the bitstream that will implement the desired functionality.

DMA controllers are responsible for handling I/O operations instead of the CPU. While a DMA transfer takes places, the CPU can handle other operations.

4.3.1. Block Diagram Design and Synthesis

In order to create the necessary bitstreams, we need to create the first block design with 2 HLS IP blocks of our choice and then synthesize it. It is best to choose the most resource-demanding HLS IP modules as the initial reconfigurable modules to implement since this makes it easier to floorplan the design and ensure latter modules fit adequately inside the marked partition. In our case, the dimer distribution HLS IP is the first that was synthesized.

After creating the initial hardware platform project with Vivado 2017.4, we selected the 3 HLS IP blocks to be available in the repository.

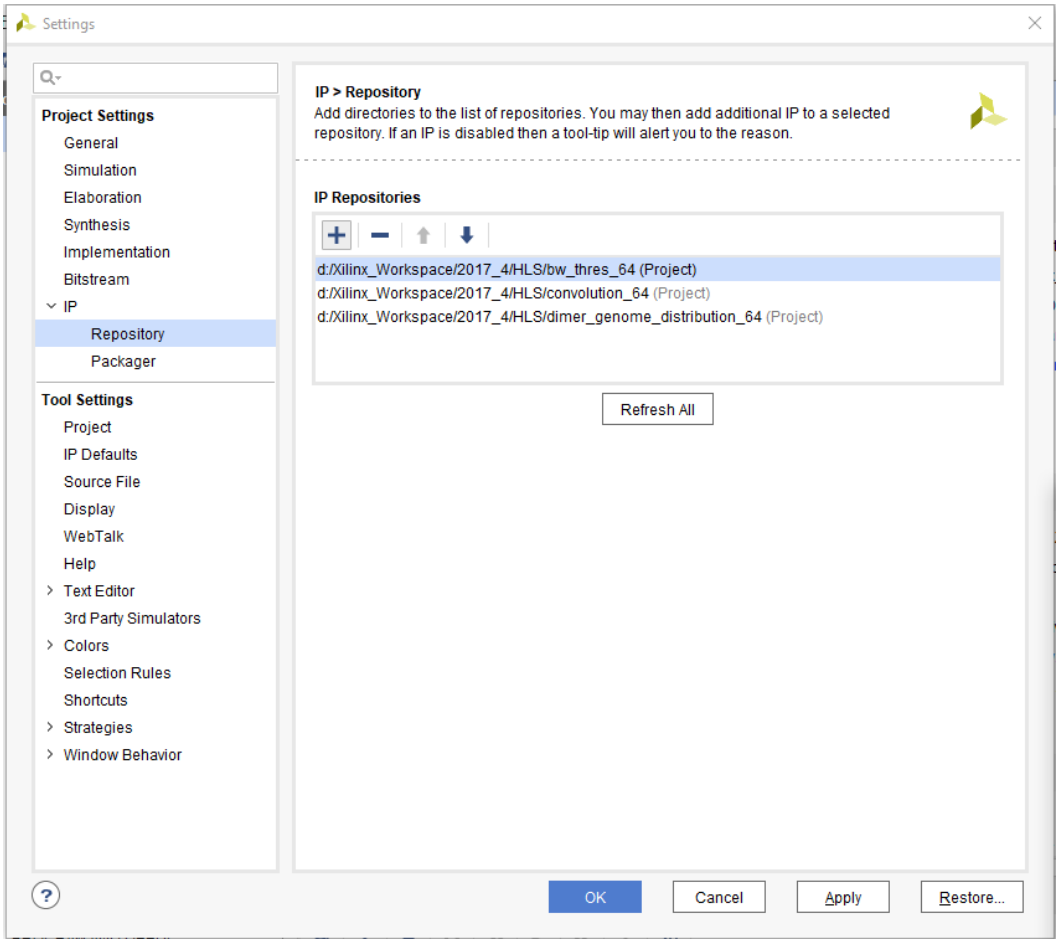


Figure 22 – Add directories of exported HLS IPs on the Vivado project

After adding the IPs in the repository, the block diagram was designed with the 2 initial implementations of the Dimer Genome Distribution HLS IP.

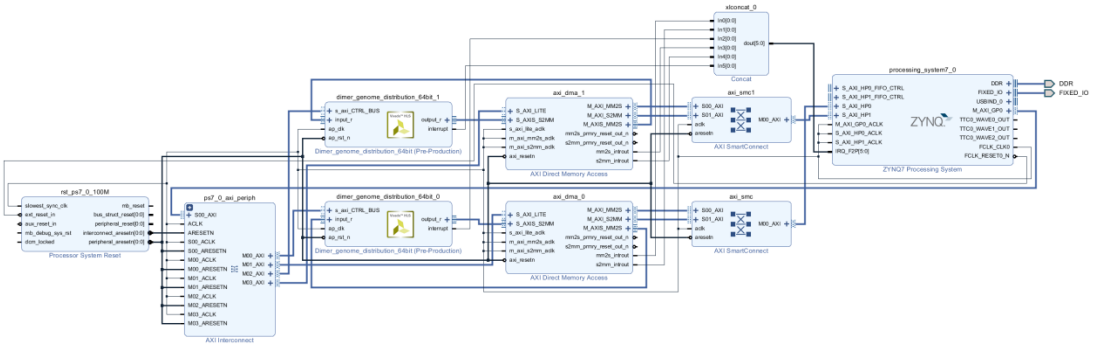


Figure 23 – Vivado block diagram of DPR platform

The only IP blocks in the design that needed editing is the 2 DMA engines (each performing transfers from each HLS IP block) and the PS7 IP.

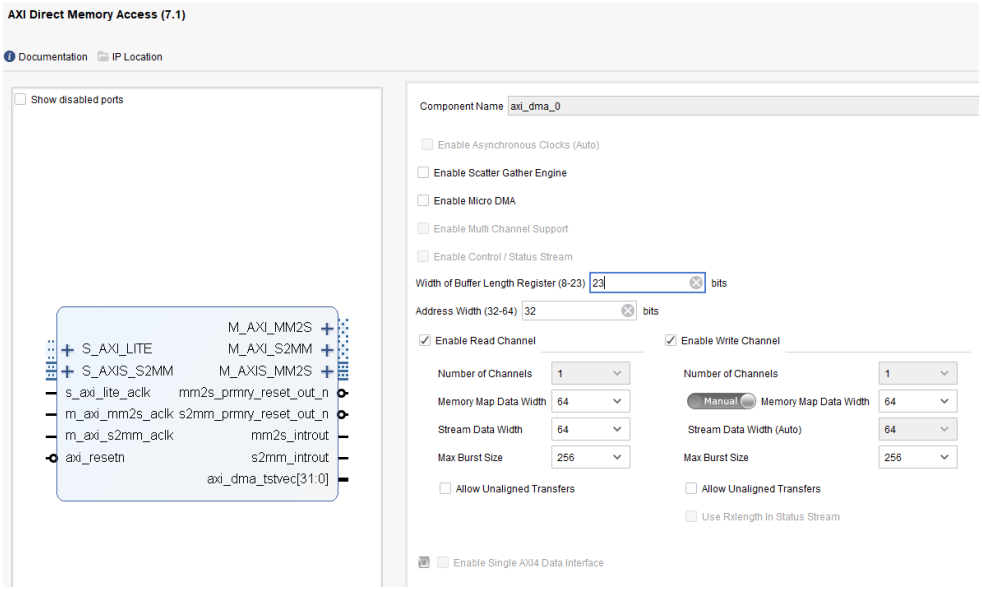


Figure 24 – AXI DMA Vivado IP Block settings

The DMA is set to allow 64-bit data transfers to allow 8 bytes per PLL clock cycle to flow through the DMA and into the IP block. The width of buffer length register setting of 23 bits means that the maximum transfer that the DMA can carry out per call in the software is 8MB.

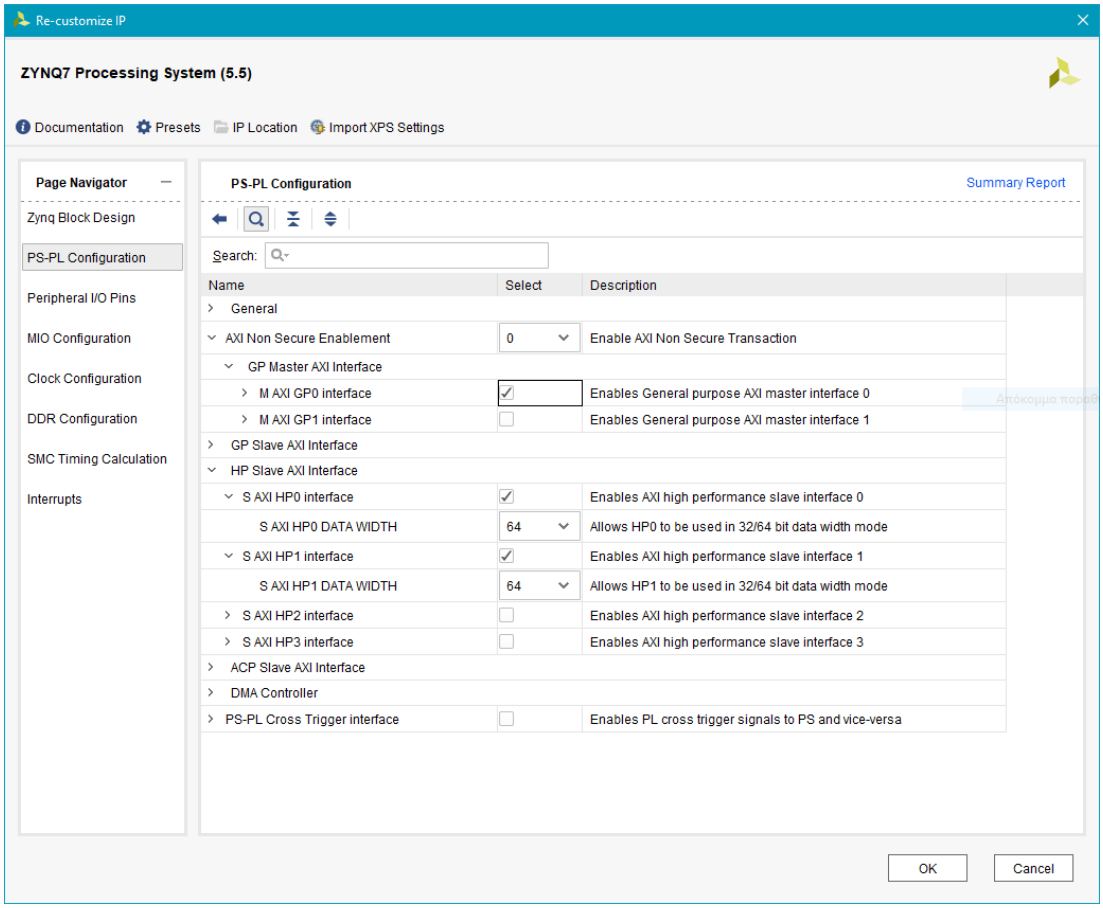


Figure 25 – PS-PL Configuration settings on ZYNQ7 Processing System IP

Enabling the GP master axi interface is necessary for propagation of scalar function arguments of HLS IPs, specifically the input and output size of data. Additionally, we make sure to enable 2 of the 4 HP Slave AXI interfaces which will be used to transfer data in and out of the 2 partitions. HP port 0 will be connected to partially reconfigurable partition 0 and HP port 1 to partition 1. It's important to set the data width to 64 bits to allow in conjunction with the 64-bit width of the DMA to flow in 8 bytes every clock cycle.

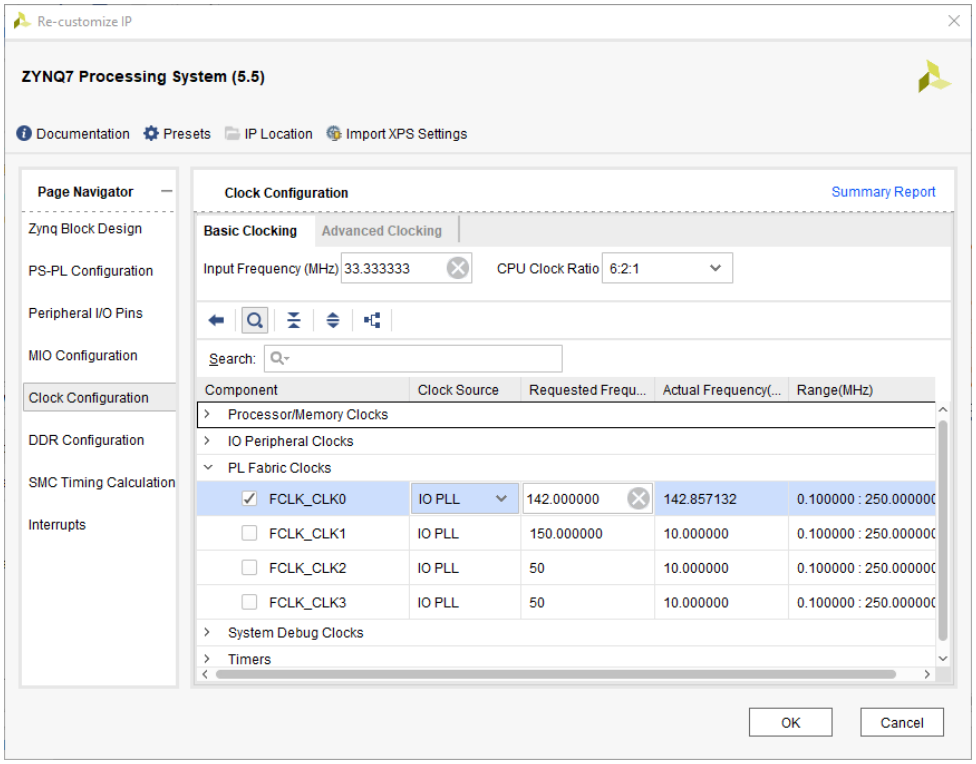


Figure 26 – PL Fabric clock settings of ZYNQ7 Processing System IP

Lastly, the clock of the PL Fabric FCLK_CLK0 is set to 142MHz, which will be set automatically by Vivado to the appropriate 7 nanosecond period – 142.857132MHz frequency.

After settings the appropriate settings in IP blocks and verifying correct configuration with the Verify functionality, the HDL wrapper for the block design was created and output products were generated in 'Out of context per IP' mode.

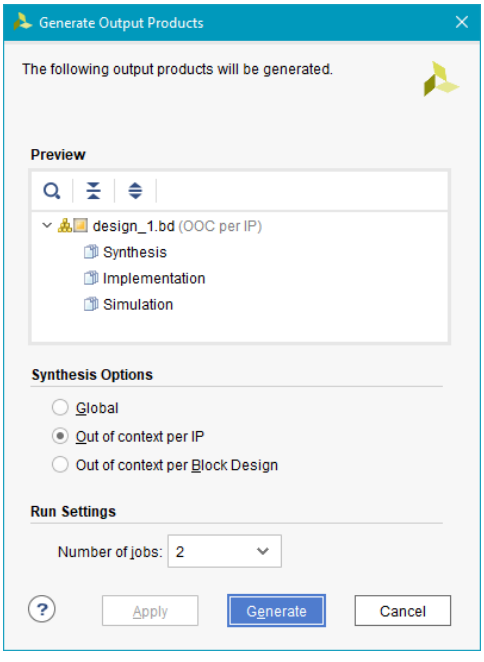


Figure 27 – Generate output products in 'OOC per IP' mode

Generating OOC per IP is important to allow Dynamic Partial Reconfiguration in the design in later stages. After generating output products the design was run through synthesis only.

After the synthesis is complete, using TCL Console and commands, we need to open the synthesized design and export the design and the cells of the Dimer Genome Distribution blocks as Design Checkpoint (.dcp) files. The cell files exported in .dcp format will later be used to be loaded into the defined Partial Region in later steps.

write_checkpoint -cell <design_path_to_cell0_dimer> <path_to_save_dcp_file>
write_checkpoint -cell <design_path_to_cell0_dimer> <path_to_save_dcp_file>
write_checkpoint <path_to_full_synth_dcp_file>

The name of the cells we want to export can be viewed in the Cell Properties panes upon selection of the cell in the Netlist pane.

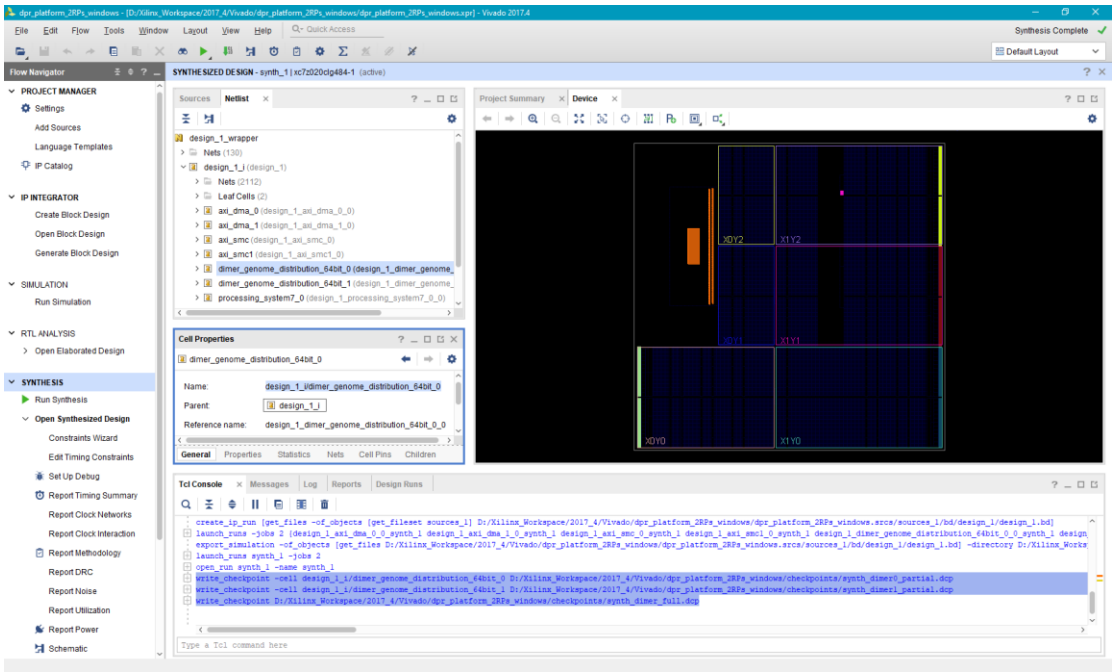


Figure 28 – Writing synthesized design and reconfigurable cells checkpoints.

After storing the DCP files, we return to the block design and replace the 2 dimer distribution blocks with any 2 copies of the implemented HLS IPs. In this case we will replace them with the BW Threshold IPs.

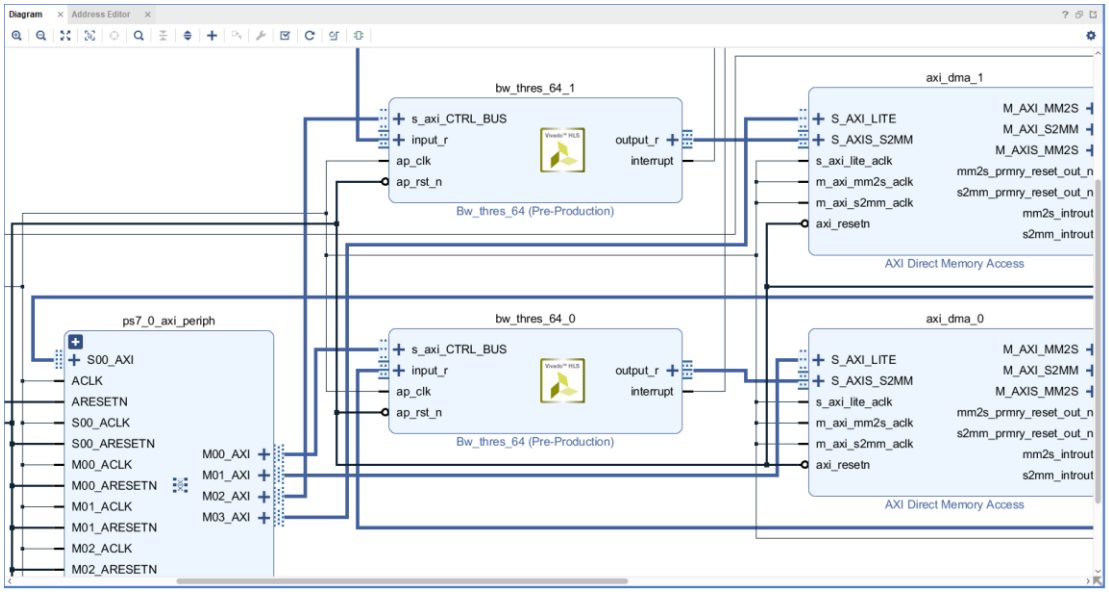


Figure 29 – BW Thresholds IPs inserted in place of Dimer Distributions HLS IPs.

After assigning the address in the Address Editor pane automatically, we verify correct connection and synthesize this design as well. After synthesis is done we open the synthesized design and press the reload design shortcut in the top of the window to view the newly synthesized design in the netlist.

Next we export to DCP files the 2 partial cells and the whole design as we did for the dimer distribution design. After exporting, we go back to the design and repeat the same process for as many modules we have implemented as we want. In this case one more remains, the Convolution Image Filter HLS IP.

4.3.2. Floorplanning and Implementation of the Hardware Design

After creating all synthesis dcp files, we set the reconfigurable cells' HD.RECONFIGURABLE property to 1. This enables Dynamic Partial Reconfiguration on the project and is non-reversible

```
set_property HD.RECONFIGURABLE 1 [get_cells <path_to_rm_cell1>]
set_property HD.RECONFIGURABLE 1 [get_cells <path_to_rm_cell2>]
```

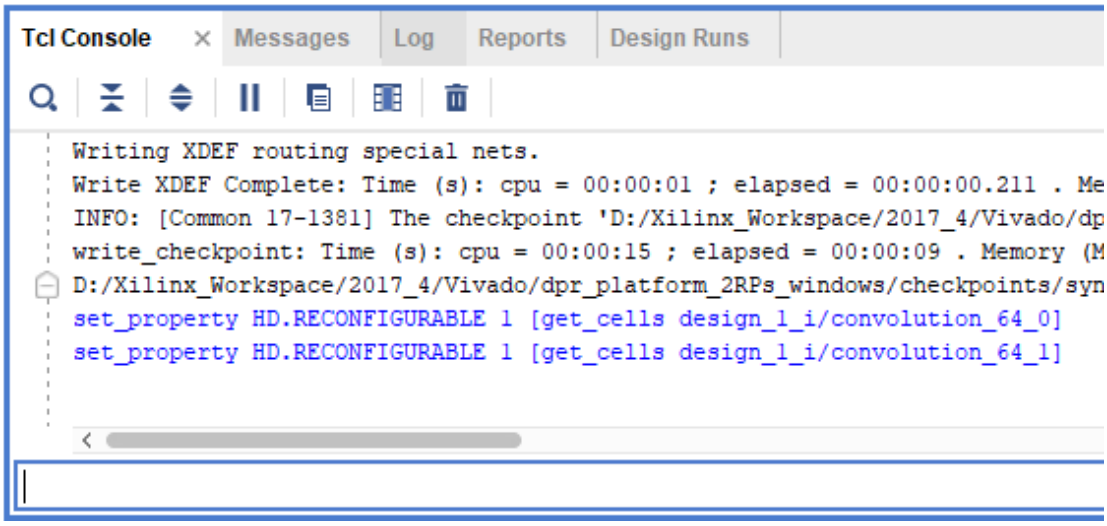


Figure 30 – Setting the HD.RECONFIGURABLE property of the design cells intended to be reconfirable

The property ‘Don’t touch’ of the 2 cells marked must be selected

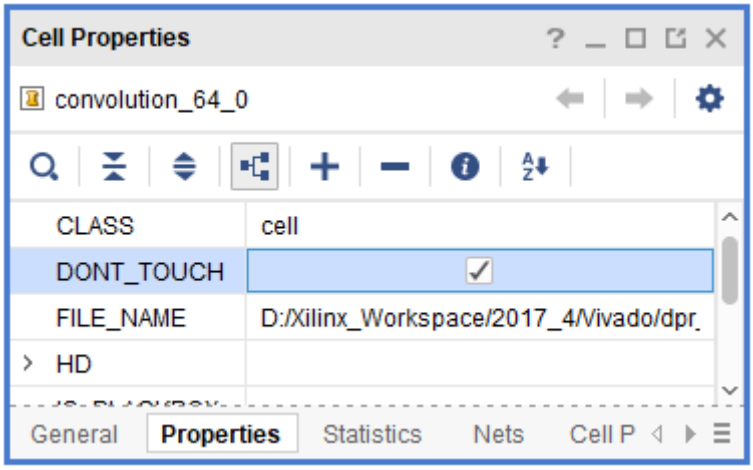


Figure 31 – Setting the DONT_TOUCH property of the reconfigurable cells

After confirming the don’t touch property is set, we move on to the floorplanning stage. In this stage we need to make sure to assign the reconfigurable modules to a partition on the FPGA fabric that contains enough resources for all other synthesized modules to be programmed on.

This involves reading the utilization reports of the synthesis stage of all generated full design checkpoint files by opening the full .dcp files generated previously, running the open_checkpoint tcl command, running the report_utilization command on the opened design checkpoint and measuring for each reconfigurable cell in each design the maximum utilization value for each of the 4 main categories of resources, Look-Up Tables, Flip-Flops, BRAM36 modules and Digital Signal Processors.

The relevant values for the current design are listed below.

Table 4 – Maximum utilization value for each category of resource for the 3 implemented algorithms

HLS IP Block	LUTs	FFs	BRAM36 Tiles	DSPs
Dimer Distribution	4254	5150	0	0
BW Threshold	406	429	0	0
Convolution	3056	5538	24	0
Maximum Value	**4254**	**5538**	**24**	**0**

After measuring this maximum value, we right click on the cells marked for reconfiguration on the opened synthesis design of the last synthesized block diagram and select Floorplanning→Draw Pblock to draw a p-block for each cell's partition, making sure each partition contains enough resources to fit all RMs. A good practice noted by Xilinx guides is to add an additional 10% of resources to allow leeway for additional routing resources.

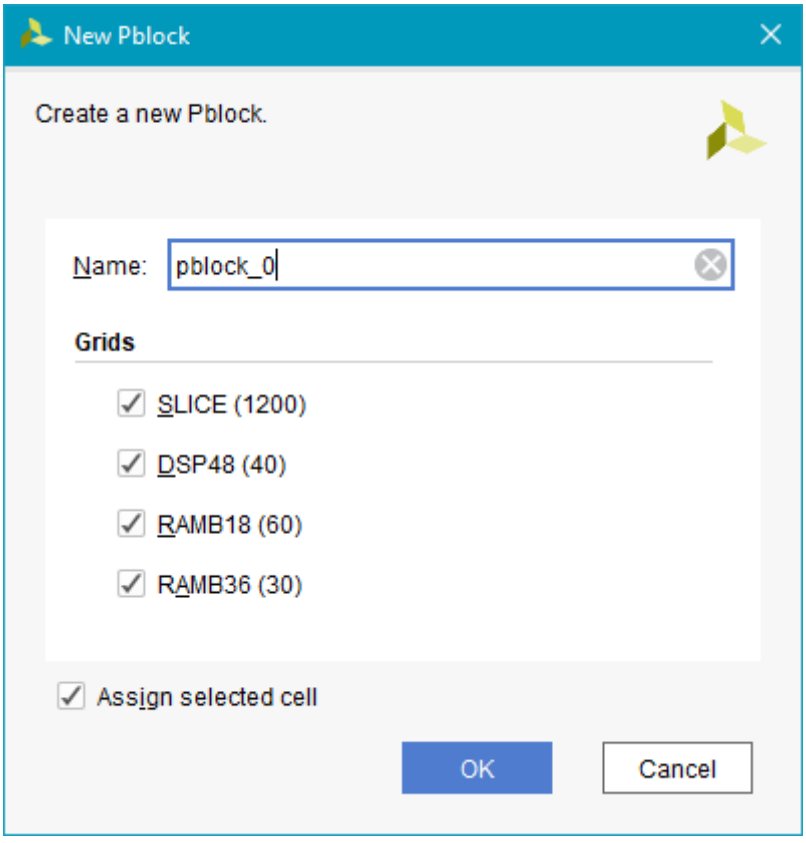


Figure 32 – Pblock creation dialog

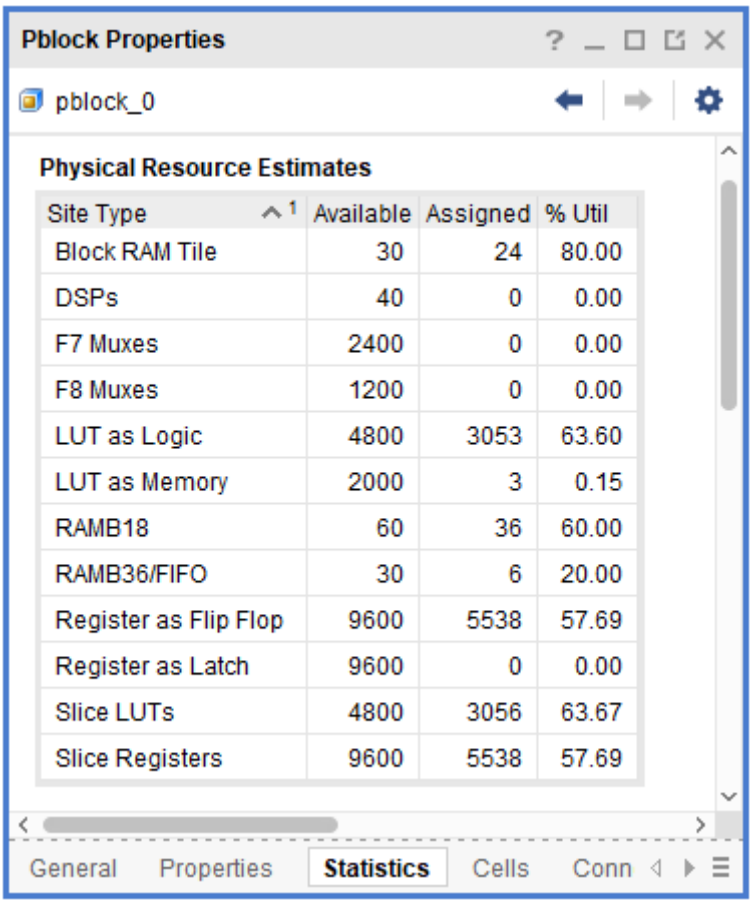


Figure 33 – Created pblock resource utilization estimates for first module.

Note the Available column in the above figure that the appropriate number of LUTs, FFs and BRAM36 tiles have been allocated for latter modules to be inserted.

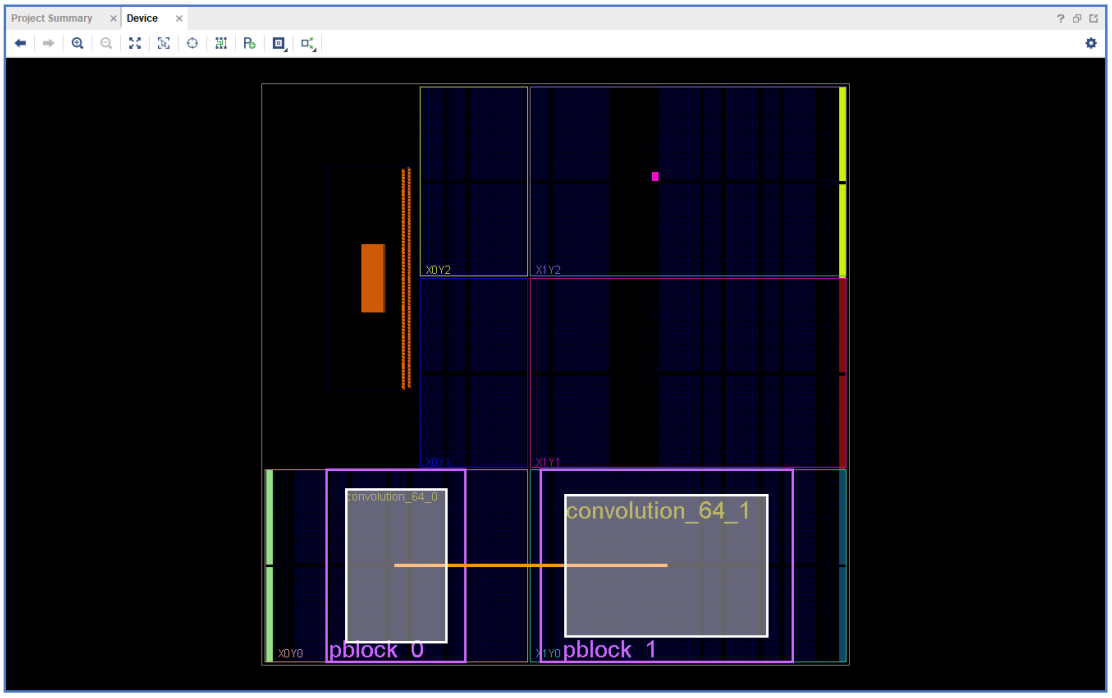


Figure 34 - Floorplanned device with 2 partial reconfiguration regions

Xilinx devices offer the capability of resetting a partial module's partition when programmed to ensure a predictable starting condition of the programmed module. In Zynq-7000 devices, this is enabled by checking the RESET_AFTER_RECONFIGURATION property of each defined partition. In order to enable this property however, the floorplanned pblock must be vertically aligned with the clock region it resides. This means that its height must be equal to the height of its encompassing clock region. The width does not matter.

After drawing pblocks for both cells, we set the properties RESET_AFTER_RECONFIGURATION to 1 and SNAPPING_MODE to ON for both partitions by clicking on a pblock in the Device view, going to the Properties pane and setting the appropriate values.

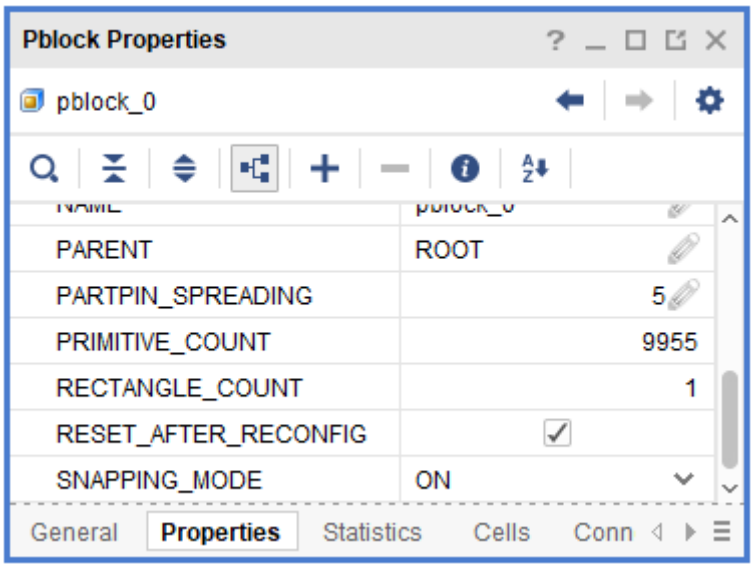


Figure 35 – Property setting of the RESET_AFTER_RECONFIG and SNAPPING_MODE properties

After setting the SNAPPING_MODE property, it's important to confirm that the floorplanned pblocks contain the necessary amount of resources to fit all modules. If resizing is needed, simply dragging an edge of the pblock should suffice.

In order to ensure correct configuration of the design up to this point we must run a Design Rule Check (DRC) report on the project. We select the Tools → Report → Report DRC…. Next we select only the Partial Reconfiguration rule subset to check in the report.

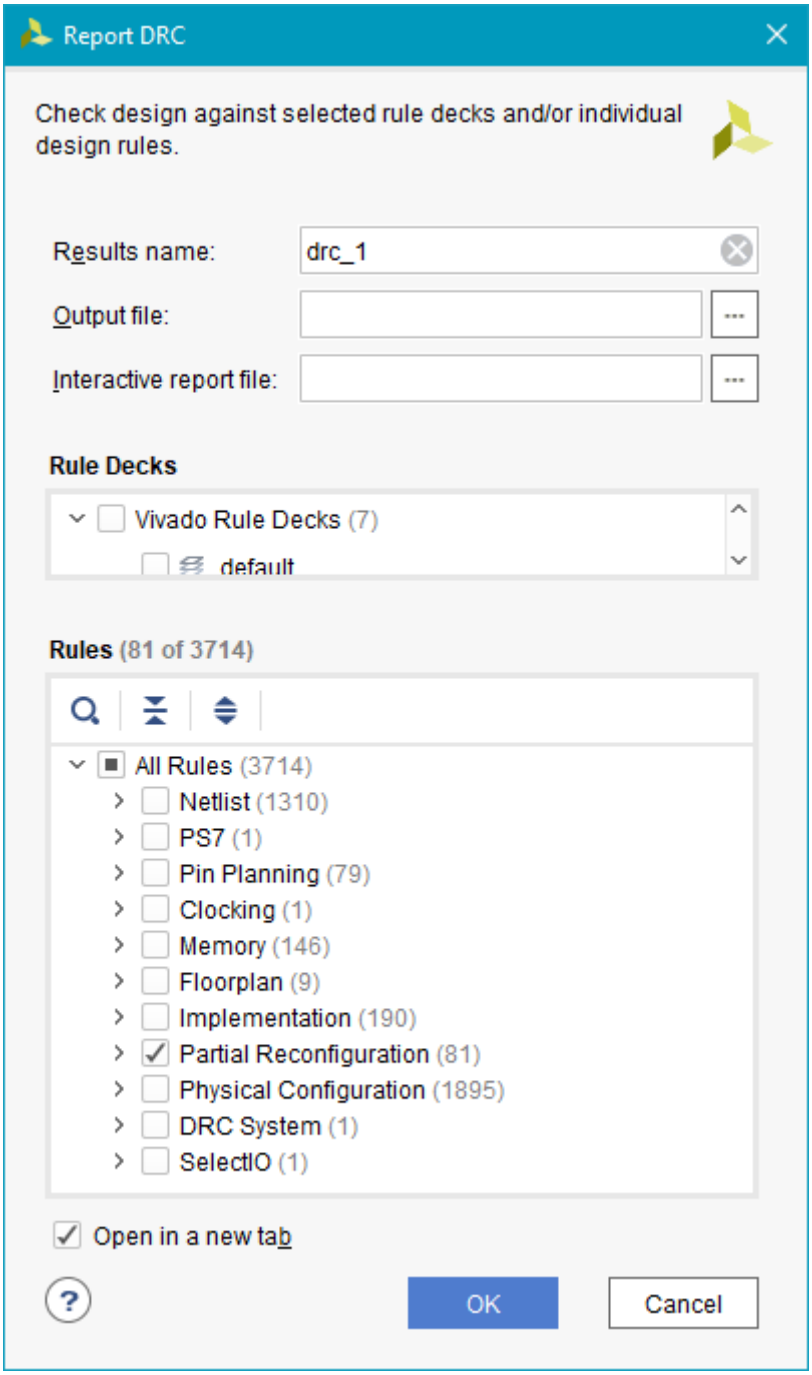


Figure 36 – DRC Rule subset selection for Partial Reconfiguration

If everything has been configured correctly, the 'No Violations Found' pop-up window should appear. Next we implement the design by running the 3 TCL commands below

```
opt_design
place_design
route_design
```

After the opt-place-route phase is over, we save a dcp file, this time in the post-implementation stage of the inserted modules that will constitute the first configuration of the DPR platform. This is done with the write_checkpoint TCL command

```
write_checkpoint <path_to_save_dcp_file_full_impl>
```

Next we must clear the 2 defined partitions by setting them as blackbox areas and lock the design routing. This stage is saved as a design checkpoint to allocate the other synthesized cells on the partial regions later and generate the blank partial bitstreams.

```
update_design -cells <path_to_rm_cell1> -black_box
update_design -cells <path_to_rm_cell2> -black_box
lock_design -level routing
write_checkpoint <dcp_save_directory>/blackbox_locked_design.dcp
```

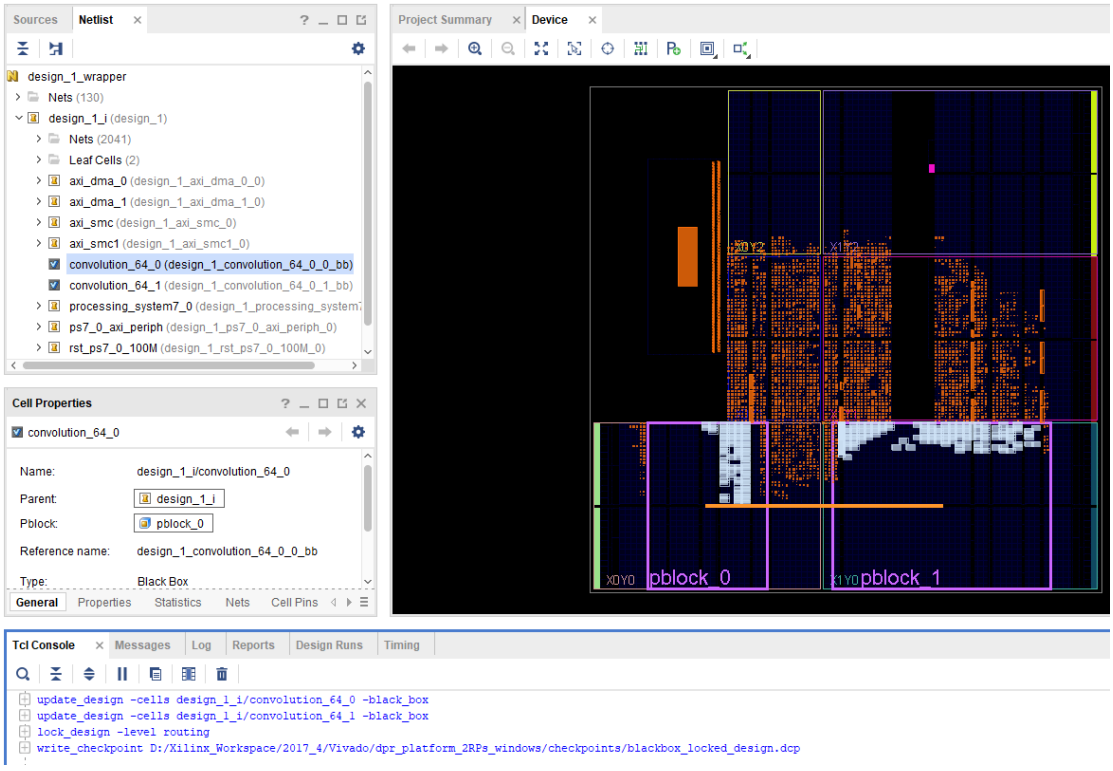


Figure 37 – Updated blackbox partitions and locked design of routing resources

At the next step we must insert the partial synth files of the remaining modules generated previously inside each corresponding partition to generate the implementation design checkpoints that will be used to generate each configuration's partial and full bitstreams.

For all the other reconfigurable modules do the following procedure using TCL commands

1. Read the checkpoints for the modules not taken care of
   read_checkpoint -cell <path_to_cell0> <path_to_partial_synth_file0>
   read_checkpoint -cell <path_to_cell1> <path_to_partial_synth_file1>
2. Do Opt-place-route process
   opt_design
   place_design
   route_design
3. Write the implemented configuration checkpoint
   write_checkpoint <path_to_dcp_file_full_impl>
4. Close the Project using the following command
   close_project
5. Read the blackbox locked checkpoint
   open_checkpoint blackbox_locked_design.dcp
6. If no more modules are left to implement, exit this loop, else go to 1)

In the opened design checkpoint of the locked blackbox design, we execute the commands below to insert LUTs tied to constant values that will ensure the outputs of the reconfigurable partition are not left floating. Additionally we will also place and route the design to create the blackbox configuration of the platform and get blank partial bitstreams.

update_design -buffer_ports -cell <path to RM cell0>
update_design -buffer_ports -cell <path to RM cell1>
place_design
route_design
write_checkpoint <path_to_impl_blackbox_design_dcp>
close_design

4.3.3. Verify Partial Reconfiguration Compatibility and Generate Bitstreams

After generating all implementation checkpoints, we must verify that all implementation configurations are replaceable on board. To do this we use the pr_verify command

pr_verify -initial <path_to_rm1_full_impl_design> -additional {<path_to_rm2_full_impl_design> <path_to_rm3_full_impl_design> <path_to_blackbox_full_impl_design>}

If the pr_verify command outputs that all dcp files are compatible with the initial defined, the next step is to generate the bitstreams.

For each implemented design checkpoint, we generate the partial and full bitstreams using the command loop below

1. Open the dcp file of a configuration
   open_checkpoint <path_to_implemented_dcp_file>
2. Write the partial and full bitstreams in .bit format
   write_bitstream –file <path_to_save_bitstream>
3. Create the .bin format of the partial .bit files. .bit files are not programmable at runtime. .bin format is the necessary format to allow runtime reconfiguration
   write_cfgmem -format BIN -interface SMAPX32 -disablebitswap -loadbit "up 0 <read_path_to_partial_bitfile>" <write_path_to_partial_bin>
4. Close the design
   close_design
5. If bitstreams are generated for all configurations exit, else go to step 1.

The last step needed is to export the hdf file of the initial design synthesized to allow development of applications on Vivado SDK. To do this, we must open the first synthesized .xpr project, create implementation and bitstream results and export the hdf file.

file mkdir <dpr_platform_directory>/dpr_platform_2_RPs.sdk

write_hwdef -force -file <dpr_platform_directory>/dpr_platform_2_RPs.sdk /system_wrapper.hdf

Tha names of the partial .bin files generated in this phase are needed in the next implementation step. The application that will read the .bin files and store them on the DDR memory must know the names of the bin files and how many regions have been defined in the hardware design.

4.4. Vivado SDK Design Workflow

After creating the hardware platform in Vivado and generating all the necessary partial and full bitstreams, we launch the Vivado SDK environment to develop the baremetal application that will handle file I/O, schedule the incoming job requests, handle the partial bitstream reconfigurations and delegate acceleration tasks to the PL.

First we create a Zynq FSBL application on the SDK to enable booting the application on an SD card. The FSBL is responsible for device initialization and programming the PL with the full bitstream.

Next we create the main application that will handle task scheduling and file I/O. In order to enable SD card reading (to load the partial .bin files on memory and the images and genome files to test the accelerators on) and writing (to write the images and statistical distribution .csv format files from the results of these acceleration tasks) we include the XilFFS library. The XilFFS library is a manufacturer-specific implementation of Fat File System drivers made by Xilinx that allows devices to read and write on non-volatile memory media formatted as FAT32.

After developing the application and building the FSBL and application binaries we use the bootgen tool [53] developed by Xilinx and integrated in Vivado SDK to create the bootable image that will be transferred to the SD card and used to boot the Zedboard. The STB Image library was used to enable reading and writing of grayscale images [54].

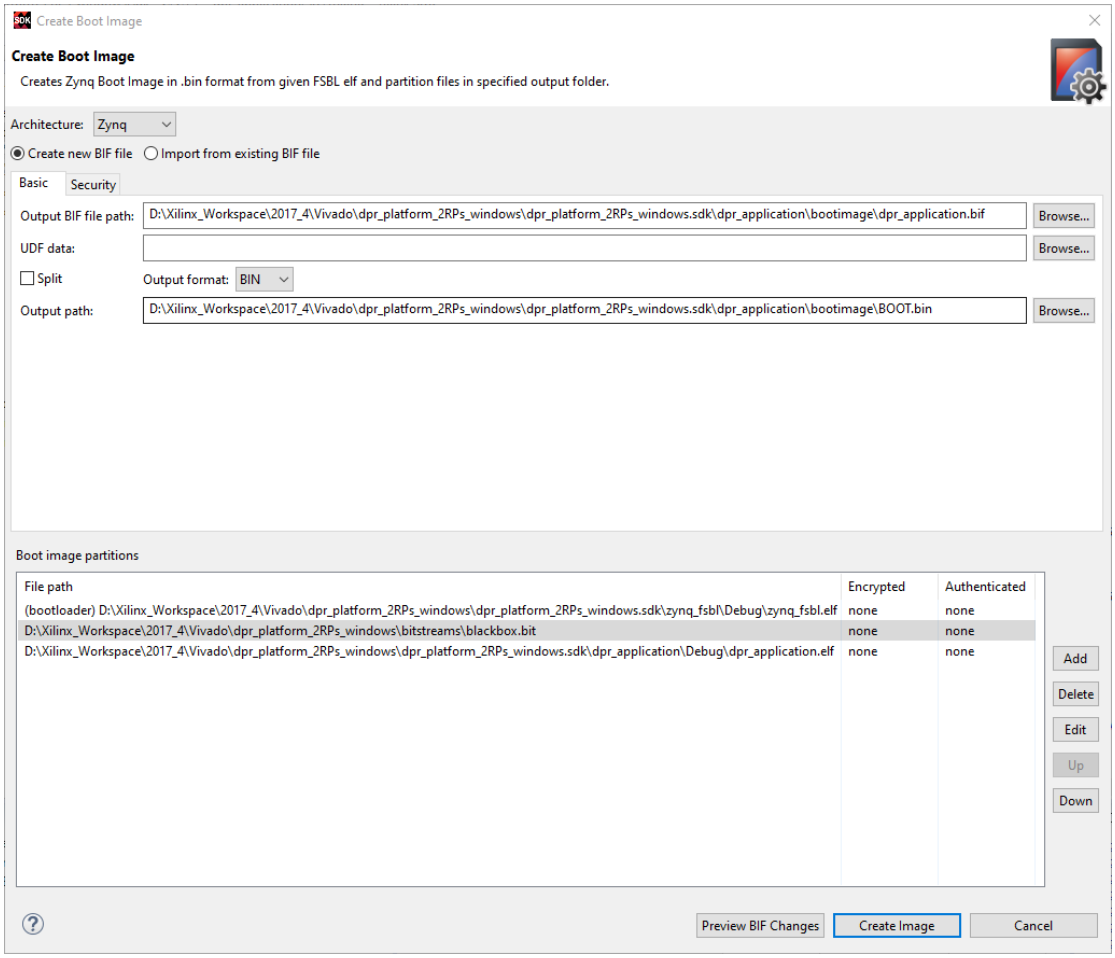


Figure 38 – Create boot image dialog box.

4.5. Power Analysis Methodology

As mentioned previously, there are several methods available for measuring the power consumption of electronic devices and computing platforms. In this work we measured power consumption of the developed FPGA platform on the Zedboard using an external current sensor, specifically the INA219 current measurement sensor [55].

INA219 sensor can measure DC in the range of +-3.2 Amps with a precision of 1%. It also comes equipped with a 12-bit resolution ADC. The resolution means that the INA219 can detect current changes of 0.8 milliamperes.

Preliminary power measurements with the ISNS20 Pmod [56] indicated that average current drawn doesn't exceed 340-350 milliamperes. INA219 is equipped with a software-configurable internal gain. The gain settings allow higher current measurement resolution at the cost of a smaller range of maximum current that we can measure. The table below shows how resolution and current measurement range change depending on the setting of the internal gain.

Table 5 – INA219 internal gain configurations.

Internal Gain Configuration	Current measurement range (in milliamps)	Current measurement resolution (in milliamps)
Div1	+-3200	0.8
Div2	+-1600	0.4
Div4	+-800	0.2
Div8	+-400	0.1

Since the maximum current measured with the ISNS20 pmod was measured to be at most 350 milliamps, we set the internal gain at div8 to allow for the maximum resolution that the INA219 can offer while remaining within the range of current drawn from the applications running on the Zedboard.

The INA219 utilizes the I2C interface to transfer measured values to a microcontroller. A second Zedboard was utilized to develop and operate a platform capable of reading and outputting measured values from the INA219 sensor module. This was done to ensure zero power consumption overhead as incurred from the operation of the INA219 was included in the measurements.

To facilitate differentiating between the Zedboard platform running the Partial Reconfiguration design and the Zedboard platform running the INA219 measurement design, we will refer to them as «Zedboard PR platform» and «Zedboard INA219 platform» respectively. The diagram below shows the connection setup of the 2 boards for measuring power drawn from the Zedboard PR platform.

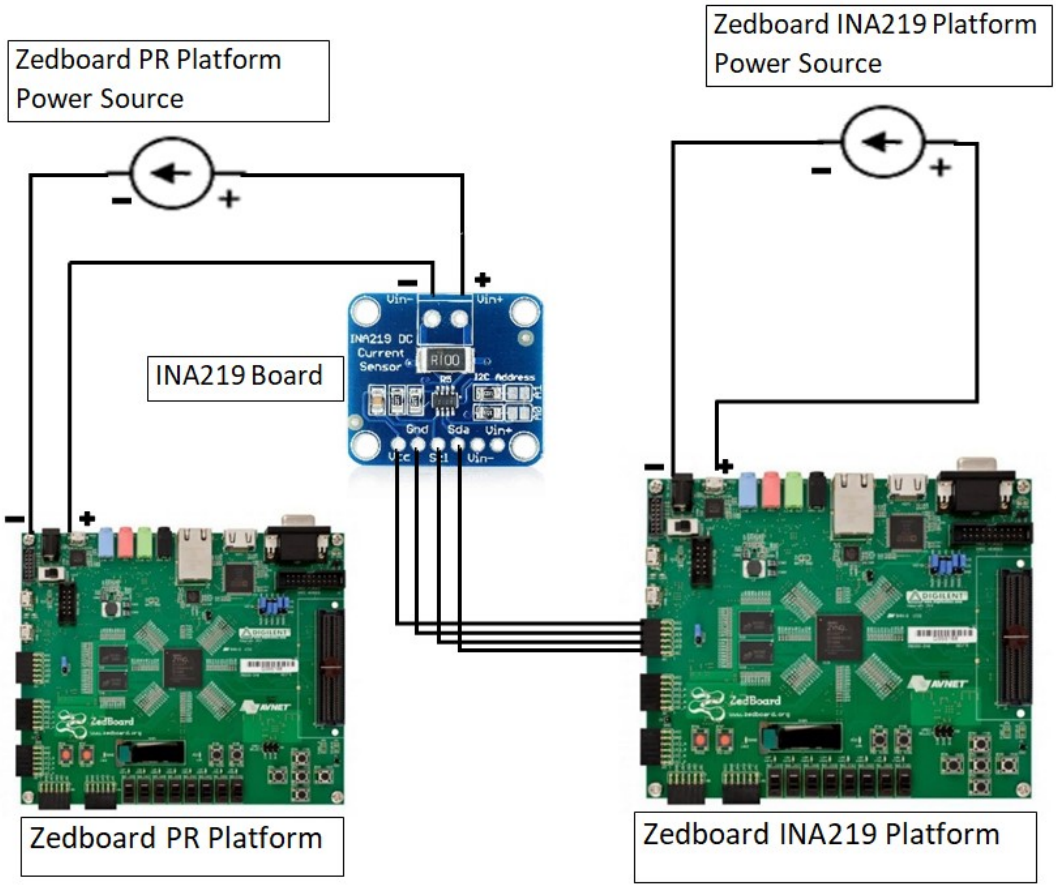


Figure 39 – Experimental setup for measuring power consumption of the developed FPGA platform.

4.6. DPR-Aware Task Scheduler Implementation

Although FPGA logic usually consumes less energy than equivalent x86_64 implementations, in order to minimize power consumption of the platform, we need to make sure to program partitions that are not used with blank partial bitstreams.

Additionally, in a cloud computing environment an FPGA could potentially accelerate a broad range of tasks, which in turn means a large number of programming bitstreams featuring all kinds of accelerators in all kinds of heterogeneous or multicore configurations, making it cumbersome to schedule them in the platform. Implementation of a task scheduler that efficiently reprograms parts of the FPGA chip on-demand while leaving the rest of the chip to function uninterrupted is necessary to increase efficiency of the platform and provide acceptable QoS.

Dynamic Partial Reconfiguration offers the capability to program a partition with a blank bitstream, a 'blackbox' Reconfigurable Module (which is termed 'greybox' in Xilinx devices), in order to reduce power consumption when not utilizing a RP. This is useful to achieve high overall power efficiency in our system and reduce idle power consumption; however there is a downside to this.

The ARM CPU utilizes the PCAP interface to program the Programmable Logic at runtime. Reprogramming requires a non-trivial amount of time to complete.

Whether we want to program a blank bitstream to reduce idle power consumption of the Programmable Logic, or we want to program a reconfigurable module to offer an acceleration service, this energy overhead must be taken into account and necessary actions need to be taken from the developer to ensure that the chance that a reconfiguration takes place is as small as possible.

From the aforementioned, the following questions are formulated that are important to answer in order to reap the benefits of partial reconfiguration in the domain of power saving.

1. What is the minimum amount of time a reconfigurable partition should retain a programmed blank bitstream in order to save more energy than the energy incurred to program it on the FPGA?
2. Given an acceleration task that has been completed by a programmed reconfigurable module on a reconfigurable region, what is the expected average time that the same task might be requested again?

In order to answer the first question we must be able to measure 2 different energy dissipation values. The first is the energy consumed by the FPGA device while reconfiguration takes place. This value is given by the equation

Equation 5 – Energy overhead of programming a partial bitstream to a reconfigurable region

$$E_{rcnfg} = P_{rcnfg} * T_{rcnfg}$$

P_{rcnfg} is the power draw of the platform during reconfiguration

T_{rcnfg} is the time needed to reconfigure a partition.

P_{rcnfg} in this work is taken from measuring the wattage of the Zedboard while it is reconfiguring a partition using the INA219 sensor.

The second energy dissipation value must be given by finding the energy difference of the energy dissipated while the reconfigurable region is loaded with a partial bitstream of an implemented algorithm while in idle mode

Equation 6 – Idle function module energy dissipation equation

$$E_{moduleIdle} = P_{funcModuleIdle} * T_{idle}$$

and while the same region is loaded with a blank bitstream.

Equation 7 – Idle blackbox module energy dissipation equation

$$E_{blank} = P_{blankModuleIdle} * T_{idle}$$

E_{blank} is the energy dissipated by the Zedboard, measured in Joules, while it is programmed with a blank bitstream.

$P_{blankModule}$ is the power consumption of the Zedboard, measured in Watts, while programmed with a blank bitstream.

T_{idle} is the time period in seconds that a region operates in idle mode.

$E_{moduleIdle}$ is the energy dissipation of a region loaded with a specific function reconfigurable module

$P_{rcnfgModule}$ is the power drawn from the logic residing in the programmed partial region

After finding the above 2 energy values, the difference energy value needs to be estimated to deduce the energy savings that the blank bitstream has incurred, since a blank bitstream will almost always draw less power than an equivalent area functioning module.

Equation 8 – Energy savings incurred from programming a blank bitstream over a functioning module

$$E_{savePartial} = E_{moduleIdle} - E_{blank}$$

The following comparison must be true in order to incur energy savings when programming blank partial bitstreams in a DPR platform.

Equation 9 – Equation to check if programming a blank bitstream incurred enough energy savings to compensate for the energy cost of programming it

$$E_{savePartial} > E_{rcnfg}$$

From the above equations, it becomes apparent that we need to maximize the $E_{savePartial}$ value and the main method that we can do this is by making sure that the expected T_{idle} time span that a region is programmed with a blank bitstream is as long as possible, or at the very least long enough to incur power savings.

Calculating E_{rcnfg} values is trivial and can be calculated by measuring time needed to reprogram a partition and the wattage of the system while reconfiguring. Calculating $E_{savePartial}$ on the other hand is less straightforward since it requires knowing the power consumption of the PL while it is programmed with a blank bitstream and while it is programmed with a function module and operating in idle mode.

Additionally, if we reprogrammed a partition with a blank bitstream every time a task was completed, the computational overhead incurred due to the partition reconfiguration would severely undermine the efficiency of the platform in carrying out data-intensive workloads. As such, reprogramming a partition with a blank bitstream immediately after completion of a task should be done only if said task is called sparsely.

The graph below shows the computation overhead of reconfiguring a partition with a module and carrying out the computation on 2 different test cases. One is applying a convolution filter on a 1920x1080 grayscale image and the other is calculating the dimer genome distribution of E. Coli.

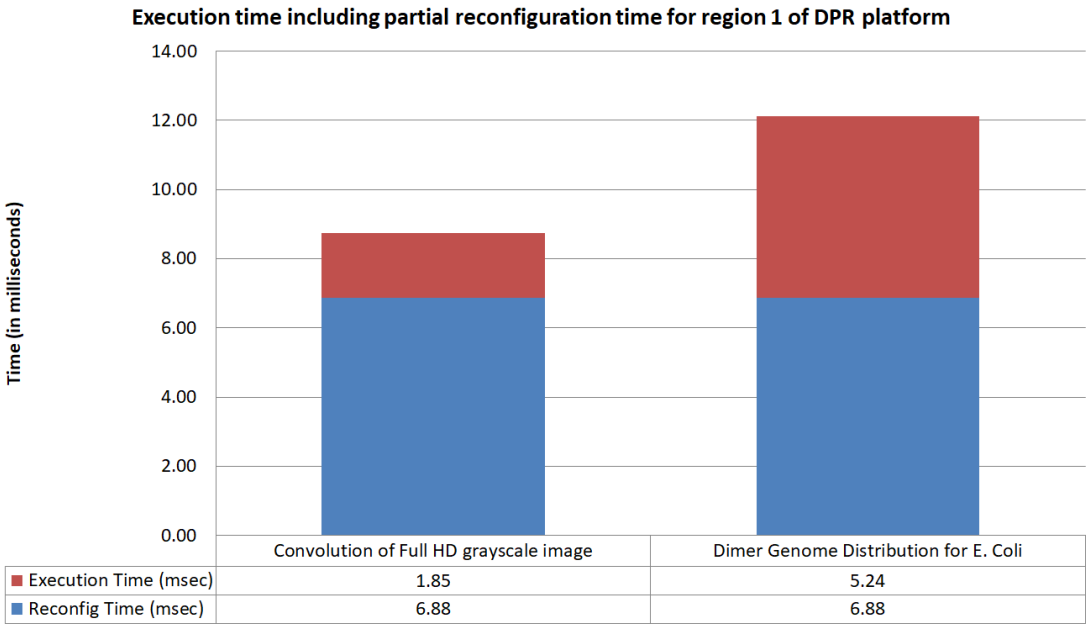


Figure 40 – Computational overhead of Dynamic Partial Reconfiguration.

Convolving a 1920x1080 grayscale image takes 1.8 msec while configuring a partition with the partial bitstream that handles convolution takes almost 4 times longer. The need for an intelligent scheduling becomes evident.

In light of the above, a task scheduler was implemented. This scheduler is responsible for monitoring which jobs are requested and effectively handling FPGA and memory resources to complete these jobs. For example, if a task (e.g. applying a Black and White threshold of 120 on a 512x512 grayscale image) is requested, the scheduler must check to see if a partition is already programmed with the module handling the requested task and if idle, simply delegates the service to this module.

Additionally, if a task is requested and no partition is already loaded with the module handling it, the partition with the module that was Least Recently Used (LRU) is loaded with the requested partial bitstream. This is akin to the LRU replacement policy used in cache memory.

Below is a generic flow diagram showcasing the scenario where a user requests processing of data for a specific task.

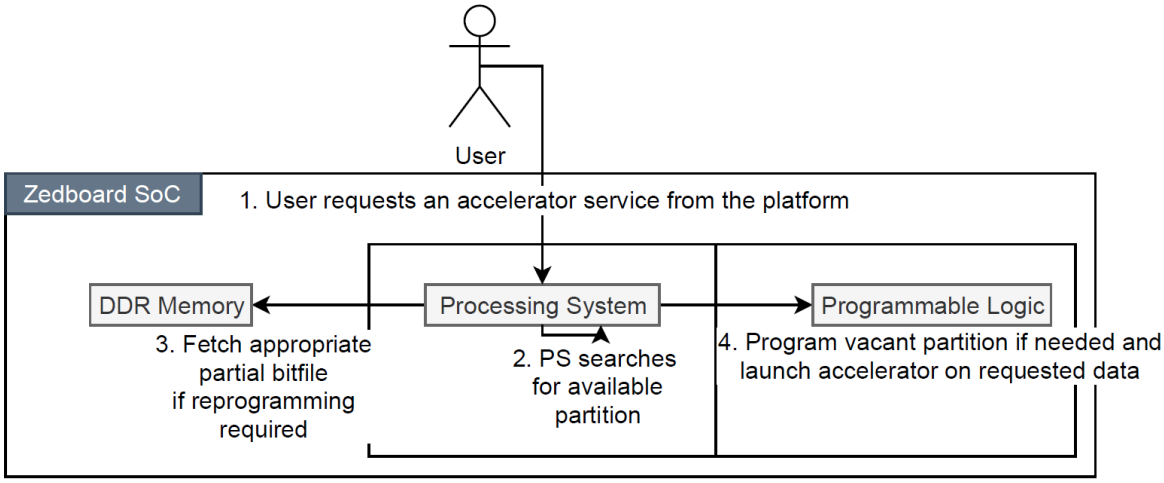


Figure 41 – Flow diagram for launching an acceleration task for requested data on the platform

In step 1, the user selects the input data (such as an image or a txt file) and the algorithm to execute on the input data. The system has been design to automatically resolve whether the input data and the requested task are compatible and an appropriate error message is issued if they are not (e.g., can't run a Black and White Threshold algorithm on a .txt file)

In step 2, the Processing System (PS) is responsible for deducing on which partition to run the selected task on. First, the PS checks all RPs to deduce whether a requested function/RM is already programmed in a RP, either from system startup or from a previous task allocation.

1. If a RP is programmed with the requested module, the selected task launches on this RP.
2. If no partition is loaded with the requested module, the least recently used partition is chosen to be programmed with the module.

Step 3 is optional and is only called if no partition is programmed with the requested module. Depending on which partition we used least recently, the appropriate partition binary file is fetched and programmed.

Finally, in step 4, the bitstream is loaded (if fetched from DDR) and the algorithm is offloaded to the PL.

During execution, the scheduler monitors how often each service has been called.

1. If a service is called and its average interval during its last 5 calls is smaller than the minimum amount of time that the blank bitstream must be retained in a partition to incur energy savings, than the scheduler will not reprogram it immediately after completing execution.
2. If a service is called and its average interval is longer than the minimum amount of time needed for the blank bitstream to incur savings, then the scheduler reprograms the partition with a blank bitstream immediately after completion.

Each accelerated service has its own timer for tracking their average call interval.

5. System Operational Metrics

In this chapter, benchmark results are presented from operation of the implemented system on various platforms. The specific metrics that will be presented on each platform are

The 3 selected algorithms were developed on 2 distinct platform configurations

1. Configuration 1 is the baremetal application that executes the 3 algorithms on the ARM A9 CPU of the Zedboard. The specifications of the platform are the following
    a. **CPU** : ARM A9 Dual-Core @667MHz
    b. **Cache**
        i. L1 32KB Data Cache (per core)
        ii. L2 512 KB cache (shared)
    c. **DDR-RAM** : 512 MB of DDR3 @533MHz
    d. **Power Consumption** : 3.82-4.02 watts (calculated wattage of the whole device, values taken from INA219 sensor readings)
2. Configuration 2 consists of the proposed FPGA system architecture that utilizes DPR techniques to allow time-multiplexed and efficient dispatching of data-intensive tasks. In this implementation, the hardware design is developed to allow partial reconfiguration of 2 regions in the FPGA whose purpose is to house any of the 3 RMs developed in Vivado HLS. Each RP requires its own partial bitstream variation of each algorithm. One synthesized and bitstream-generated RM cannot fit into any RP defined in the hardware design. A generated partial bitstream meant for partition A will not be compatible with partition B. For a given DPR-enabled platform where we want all regions marked as reconfigurable to house any of the functions developed in Vivado HLS, the equation that calculates how many different partial bitstreams need to be generated is given by the simple formula

$$N = p * m$$

Where:

N is the total number of partial bitstreams needed
p is the total number of RPs defined in the hardware platform
m is the number of different HLS IP algorithms

For each of the 2 test configuration platform and for each algorithm implementation on each platform, the following metrics will be presented

1. Computation time in milliseconds
2. Performance throughput in Mbytes/sec
3. Energy efficiency in MB processed/joule spent

4. CPU ClockCycles/byte metrics. For the ARM CPU implementation, the CPU frequency of 667 MHz will be used. For the FPGA implementation, the clocking frequency set for the PL will be used (~143MHz)

5.1. ARM Cortex A9 CPU benchmarks

In this chapter, performance and energy efficiency metrics of the ARMv7 Cortex A9 CPU embedded on the Zedboard are presented. The development environment is Vivado SDK 2017.4. The programming language used to implement the algorithms is C. Compiler optimization was set to –O3 when building the applications.

The timer used to measure runtime of each case is the Snoop Control Unit Timer (SCU Timer) embedded in Zynq family devices. SCU timer has a resolution of 3 nanoseconds.

Power efficiency was calculated from power consumption measurements taken from the INA219 current sensor.

Clock cycles per byte metrics are based on the ARM A9 CPU's clock frequency of 667 MHz.

5.1.1. Black and White Thresholding Benchmarks – ARM CPU

The table below showcases benchmark metrics for computing BW Threshold. The test data is sample data of random values in the range of [0, 255]. The sample size starts from 64 bytes and doubles until it reaches 8MB.

Power efficiency for the software implementation of the Black and White Thresholding application is calculated with a measured wattage of 3913 milliWatts.

Table 6 – BW Threshold benchmark metrics. ARM A9 CPU, -O3 optimized

Size(bytes)	Time (msec)	Throughput (in MB/sec)	Clockcycles/byte	Power Efficiency (in MB/Joule)
64	0.0004	147.64	4.31	37.73
128	0.0007	186.91	3.40	47.77
256	0.0012	196.10	3.24	50.11
512	0.0025	198.95	3.20	50.84
1024	0.0049	199.88	3.18	51.08
2048	0.0098	199.96	3.18	51.10
4096	0.0197	198.31	3.21	50.68
8192	0.0378	206.68	3.08	52.82
16384	0.0807	193.64	3.28	49.49
32768	0.1987	157.26	4.04	40.19
65536	0.4143	150.87	4.21	38.56
131072	0.8249	151.53	4.20	38.72
262144	1.6821	148.62	4.28	37.98
524288	3.5451	141.04	4.51	36.04
1048576	7.1957	138.97	4.57	35.52
2097152	14.2956	139.90	4.54	35.75
4194304	29.7955	134.25	4.74	34.31
8388608	60.0289	133.27	4.77	34.06

5.1.2. Convolution Benchmarks – ARM CPU

The table below showcases benchmark metrics for convolution. The test data is sample data of random values in the range of [0, 255]. The sample size starts from 64 bytes and doubles until it reaches 8MB. Power efficiency for the software implementation of the Image Convolution application is calculated with a measured wattage of 3857 milliWatts.

Table 7 – Convolution benchmark metrics. ARM A9 CPU, -O3 optimized

Size(bytes)	Time (msec)	Throughput (in MB/sec)	Clockcycles/byte	Power Efficiency (in MB/Joule)
64	0.0024	25.19	25.24	6.53
128	0.0054	22.68	28.04	5.88
256	0.0113	21.53	29.52	5.58
512	0.0226	21.58	29.46	5.59
1024	0.0491	19.90	31.94	5.16
2048	0.0954	20.46	31.07	5.31
4096	0.1957	19.96	31.85	5.18
8192	0.3854	20.27	31.36	5.26
16384	0.7768	20.12	31.61	5.22
32768	1.5409	20.28	31.35	5.26
65536	3.0997	20.16	31.53	5.23
131072	6.1715	20.25	31.39	5.25
262144	12.5095	19.98	31.81	5.18
524288	24.7851	20.17	31.52	5.23
1048576	49.5933	20.16	31.53	5.23
2097152	99.2031	20.16	31.54	5.23
4194304	198.4374	20.16	31.54	5.23
8388608	396.9096	20.16	31.54	5.23

Convolution using a 3x3 kernel is a more complex algorithm, requiring more computations to carry out compared to a BW Thresholding task, hence the lower throughput and efficiency measurements.

5.1.3. Dimer Genome Distribution Benchmarks – ARM CPU

The table below showcases benchmark metrics for dimer genome di stribution. The test data is sample data of random nucleobase values in the set [‘A’,‘C’,‘G’,‘T’]. The sample size starts from 64 bytes and doubles until it reaches 8MB.

Power efficiency for the software implementation of the Dimer Genome Distribution application is calculated with a measured wattage of 4012 milliWatts.

Table 8 – Dimer Genome Distribution benchmark metrics. ARM A9 CPU, -O3 optimized

Size(bytes)	Time (msec)	Throughput (in MB/sec)	Clockcycles/byte	Power Efficiency (in MB/Joule)
64	0.0015	40.70	15.62	10.14
128	0.0029	41.76	15.22	10.41
256	0.0058	41.95	15.16	10.46
512	0.0116	42.11	15.10	10.50
1024	0.0231	42.19	15.07	10.52
2048	0.0463	42.22	15.06	10.52
4096	0.0925	42.25	15.05	10.53
8192	0.1849	42.25	15.05	10.53
16384	0.3698	42.25	15.05	10.53
32768	0.7435	42.03	15.13	10.48
65536	1.5360	40.69	15.62	10.14
131072	3.1016	40.30	15.78	10.05
262144	6.2099	40.26	15.79	10.03
524288	12.4212	40.25	15.79	10.03
1048576	24.8944	40.17	15.83	10.01
2097152	49.8139	40.15	15.84	10.01
4194304	99.6344	40.15	15.84	10.01
8388608	199.2731	40.15	15.84	10.01

5.2. DPR-Enabled FPGA Design benchmarks

In this chapter, performance and energy efficiency metrics of the FPGA coprocessors in the Zynq 7020 chip embedded on the Zedboard are presented. The development environment is Vivado SDK 2017.4. The programming language used to implement the algorithms in Vivado HLS is C++.

Power efficiency was calculated from readings taken from the INA219 current sensor.

Because the CPU must read the images/genome sequences to process and write the results as .png or .csv files respectively and because the PS-PL data propagates through the HP ports which are not cache-coherent, cache flushing and invalidation must be used in the test runs. The time and energy expedited for file I/O and cache flushing/invalidating is not taken into account in the following tests.

Clock cycles per byte metrics are based on the FCLK_CLK0 used in the Vivado Hardware design and set to operate at 142.85 MHz.

5.2.1. Black and White Thresholding Benchmarks - FPGA

The table below showcases benchmark metrics for computing BW Threshold. The test data is sample data of random values in the range of [0, 255]. The sample size starts from 64 bytes and doubles until it reaches 8MB.

Power efficiency for the hardware implementation of the Black and White Thresholding application is calculated with a measured wattage of 4012 milliWatts.

Table 9 – BW Threshold benchmark metrics. FPGA Coprocessor

Size(bytes)	Time (msec)	Throughput (in MB/sec)	Clockcycles/byte	Power Efficiency (in MB/Joule)
64	0.0021	29.76	4.58	6.50
128	0.0022	54.56	2.50	11.91
256	0.0022	109.78	1.24	23.97
512	0.0026	191.01	0.71	41.71
1024	0.0029	337.15	0.40	73.61
2048	0.0039	497.93	0.27	108.72
4096	0.0058	677.92	0.20	148.02
8192	0.0093	840.43	0.16	183.50
16384	0.0165	945.68	0.14	206.48
32768	0.0308	1014.68	0.13	221.55
65536	0.0595	1049.89	0.13	229.23
131072	0.1168	1070.09	0.13	233.64
262144	0.2316	1079.65	0.13	235.73
524288	0.4609	1084.89	0.13	236.87
1048576	0.9197	1087.33	0.13	237.41
2097152	1.8371	1088.65	0.13	237.70
4194304	3.6722	1089.27	0.13	237.83
8388608	7.3422	1089.60	0.13	237.90

5.2.2. Convolution Benchmarks - FPGA

The table below showcases benchmark metrics for convolution. The test data is sample data of random values in the range of [0, 255]. The sample size starts from 64 bytes and doubles until it reaches 8MB.

Table 10 – Convolution benchmark metrics. FPGA coprocessor

Size(bytes)	Time (msec)	Throughput (in MB/sec)	Clockcycles/byte	Power Efficiency (in MB/Joule)
64	0.018	3.354	40.618	0.70
128	0.018	6.698	20.341	1.40
256	0.018	13.276	10.262	2.77
512	0.019	26.302	5.180	5.49
1024	0.019	51.325	2.654	10.71
2048	0.020	97.641	1.395	20.38
4096	0.022	179.494	0.759	37.47
8192	0.025	308.829	0.441	64.47
16384	0.032	480.771	0.283	100.37
32768	0.047	666.756	0.204	139.20
65536	0.076	827.363	0.165	172.73
131072	0.133	940.793	0.145	196.41
262144	0.247	1010.125	0.135	210.88
524288	0.477	1048.448	0.130	218.88
1048576	0.936	1068.840	0.127	223.14
2097152	1.853	1079.236	0.126	225.31
4194304	3.688	1084.547	0.126	226.42
8388608	7.358	1087.238	0.125	226.98

FPGA implementations tend to be more deterministic than conventional CU architectures, hence the similar to the BW Threshold performance

5.2.3. Dimer Genome Distribution Benchmarks - FPGA

The table below showcases benchmark metrics for dimer genome distribution. The test data is sample data of random nucleobase values in the set [‘A’,‘C’,‘G’,‘T’]. The sample size starts from 64 bytes and doubles until it reaches 8MB.

Table 11 – Dimer Genome Distribution benchmark metrics. FPGA coprocessor

Size(bytes)	Time (msec)	Throughput (in MB/sec)	Clockcycles/byte	Power Efficiency (in MB/Joule)
64	0.010	6.051	22.515	1.48
128	0.010	12.047	11.309	2.94
256	0.010	23.875	5.706	5.82
512	0.010	46.579	2.925	11.36
1024	0.011	89.355	1.525	21.79
2048	0.012	165.959	0.821	40.48
4096	0.014	286.986	0.475	70.00
8192	0.017	453.790	0.300	110.68
16384	0.024	642.703	0.212	156.76
32768	0.039	808.540	0.169	197.20
65536	0.067	928.629	0.147	226.49
131072	0.125	1002.316	0.136	244.47
262144	0.239	1044.298	0.130	254.71
524288	0.469	1066.672	0.128	260.16
1048576	0.928	1078.158	0.126	262.97
2097152	1.845	1083.998	0.126	264.39
4194304	3.680	1086.967	0.125	265.11
8388608	7.350	1088.440	0.125	265.47

5.3. Partial Reconfiguration Energy Overhead

In this chapter, we will present measurements from reconfiguration time and energy overheads incurred when loading a partial bitstream on a RP.

In the graph below we can see the time and energy cost of reconfiguring each of the 2 partial regions defined in this work.

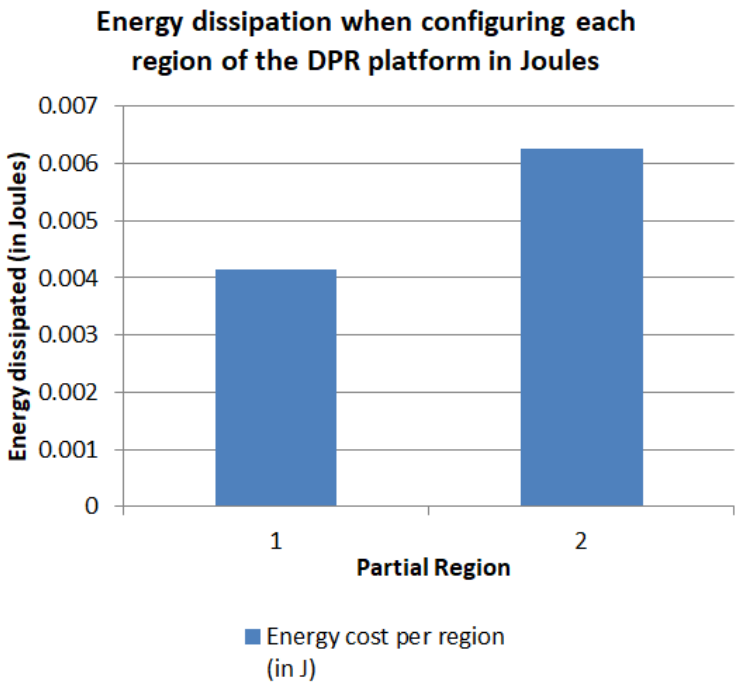
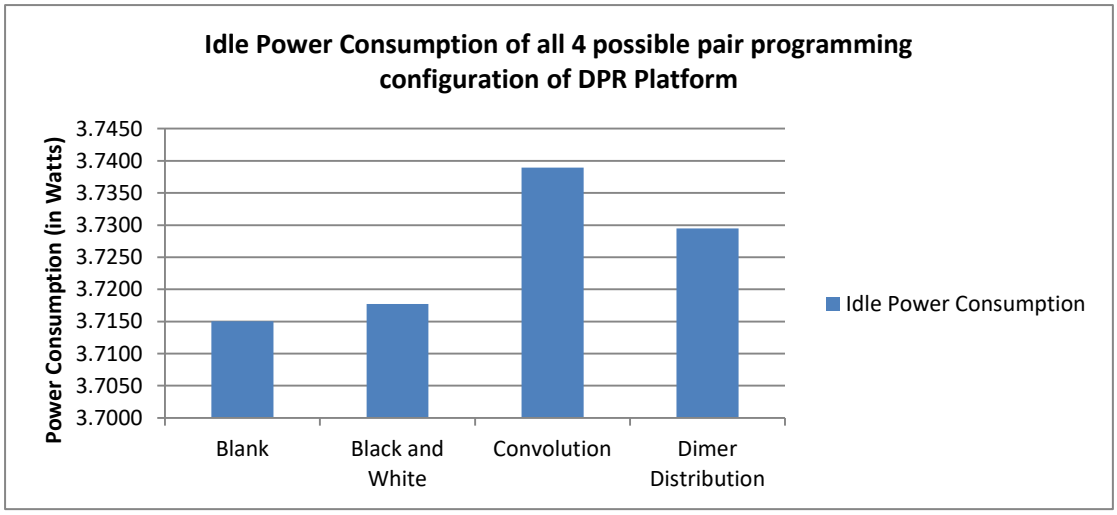


Figure 42 – Energy cost of programming a partial region

From the graph above, it becomes evident that each region has each own cost of reprogramming it. It is not easy to define partial regions in a DPR platform design that house the necessary resources inside and simultaneously have the same size when generated as partial bitstreams.



From analysis done on the power drawn from the Image Convolution module and the Blackbox module when they are programmed on region 1 of the DPR platform we have concluded that the device draws approximately 12.7 milliWatts less when programmed with the blackbox module. As such from Equation 8 and Equation 9 we derive that $E_{savePartial}$ must be higher than 4.13 milliJoules. Solving for T_{idle} to find the minimum time needed

$$T_{idle} = \frac{E_{rcnfg\_region0}}{P_{diffIdle}} = \frac{0.00413}{0.0127} = 0.323\ seconds$$

Where

$E_{rcnfg\_region0}$ is the energy cost of reconfiguring region 0

$P_{diffIdle}$ is the power draw difference of the blackbox module and the convolution module.

As such, after programming region 1 with the blank partial bitstream, it should remain at least 323 milliseconds in the region in order to incur enough energy savings to compensate for its reprogramming cost. The same T_{idle} value computed for region 2 which is a larger region, incurring a larger energy overhead cost to reprogram is 490 milliseconds.

For different modules this time is different, since other modules may draw lower or higher power when in idle mode. This means that the time needed for a blank bitstream to remain in the region to compensate its reprogramming cost may be different when replacing different modules.

Additionally, different reconfigurable regions have different sizes in partial bitstreams and thus take different time spans to reprogram. This should be taken into account when dynamically reprogramming regions in an energy-aware platform.

Although results show that a partial region could remain programmed with a blank bitstream for a few hundreds of milliseconds before incurring energy savings, performance goals may indicate that we may still want to retain an idle module on a configured region for longer than this time span.

This is due to the fact that an acceleration platform needs to meet performance demands alongside energy efficiency goals and a service or user that may request the same acceleration task can utilize this module and not incur the performance overhead of reprogramming a region with that module.

As such, the option to keep a programmed module for a time longer than T_{idle} has been implemented as well. System administrators may select this mode of operation as per their requirements at any time.

6. Experimental results discussion

In this chapter the findings of the implementation in this work are presented and experimental results of the proposed system are discussed and analyzed.

6.1. Execution runtime comparison

Execution runtime of the 3 algorithms on the 3 testing platforms are presented. File I/O time is not included in the measurements.

6.1.1. Black and White Image Thresholding runtime

The graph below shows the runtime of the BW Image Thresholding algorithm in relation to the input size of the data measured in milliseconds on the 2 test platforms (ARM A9 CPU, FPGA accelerator IP). The Y axis showing the execution time is in logarithmic scale.

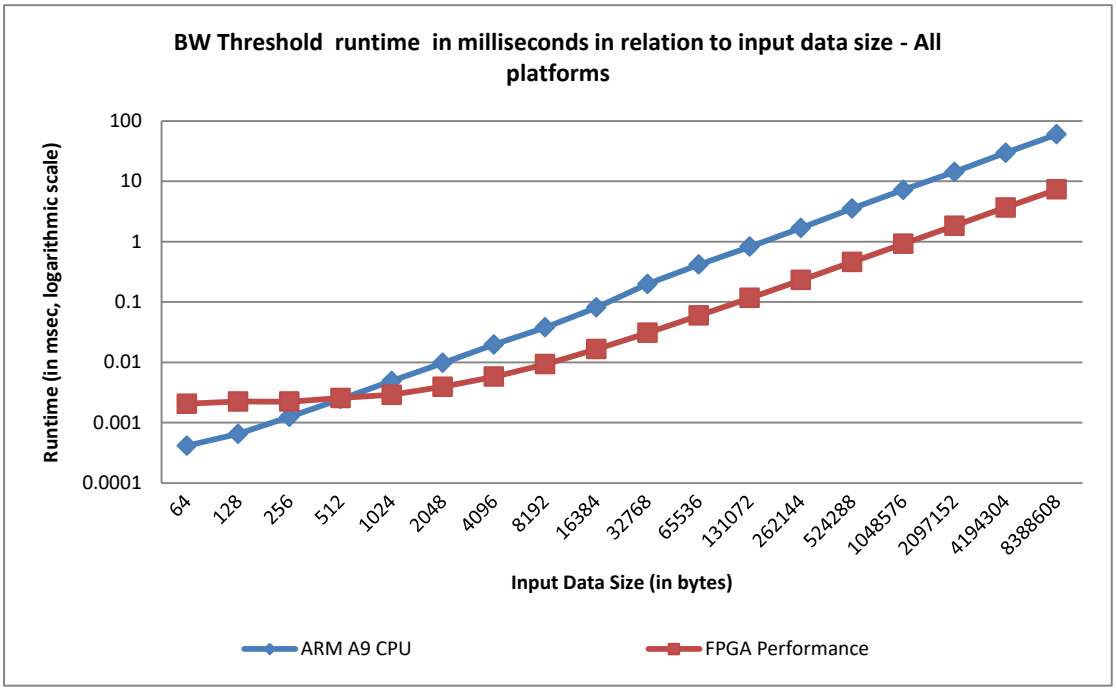


Figure 43 – BW Threshold runtime in milliseconds graph, all platforms compared, semi-logarithmic graph

The FPGA implementation shows an almost static 2 microseconds execution time for the first 4 input sizes, which is attributed attribute to the latency in transferring data from the DDR to the Programmable Logic.

As the size of the processed data increases however, this 2 microsecond latency takes up an ever smaller percentage of overall execution time. We also notice that for input sizes in the range [32KB-8MB] the FPGA coprocessor is almost 1 order of magnitude faster than the ARM CPU implementation.

6.1.2. Image Convolution runtime comparison

The graph below shows the runtime of the Image Convolution with 3x3 kernel algorithm in relation to the input size of the data measured in milliseconds on the 2 test platforms (ARM A9 CPU, FPGA accelerator IP). The Y axis showing the execution time is in logarithmic scale.

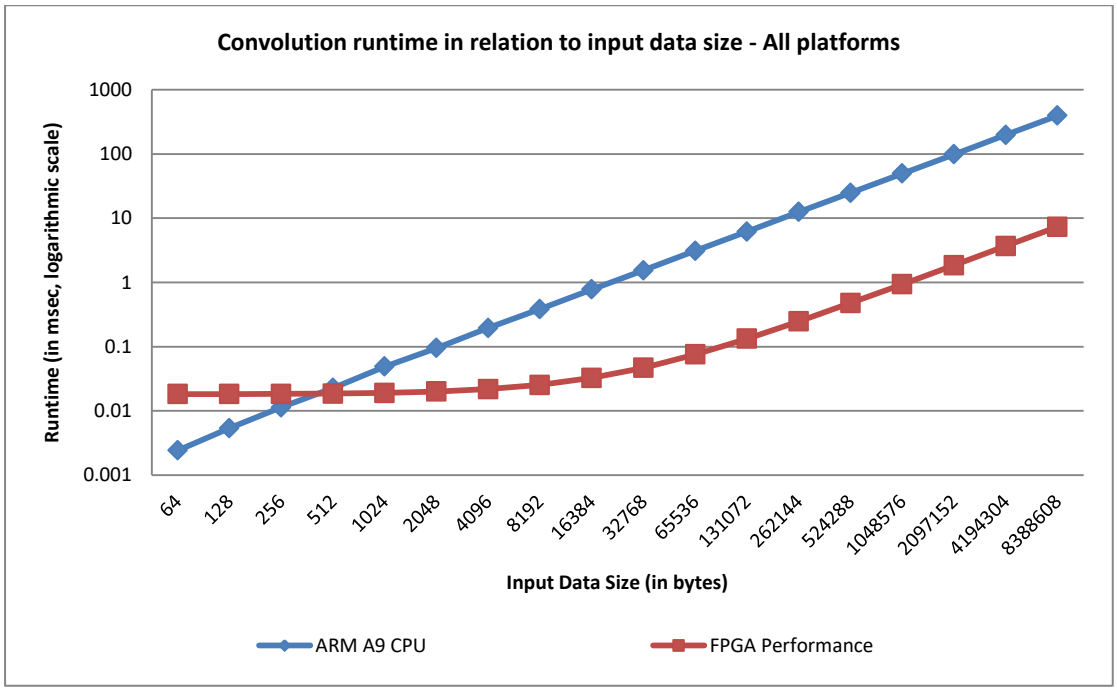


Figure 44 – Image Convolution with 3*3 Kernel runtime, all platforms compared, semi-logarithmic graph

In the case of the Image Convolution algorithm, it is clear that the FPGA implementation is much faster than the ARMv7 CPU implementation. For an input size of 2MB (almost exactly the size of a full HD grayscale image) the processing time on the FPGA platform is 1.86 milliseconds, while the ARMv7 CPU platform is 55 times slower at 99.2 milliseconds.

Similarly to the BW Image Thresholding algorithm, the Image Convolution FPGA accelerator shows a steady runtime of almost 18 microseconds for the first 5 input sizes tested due to the latency of transferring data. The additional 16 microseconds delay compared to the previous algorithm is caused from the setup of look-up tables and parsing of operational parameters that take place prior to processing the actual input image data.

6.1.3. Dimer Genome Distribution runtime comparison

The graph below shows the runtime of the Dimer Genome Distribution algorithm in relation to the input size of the data measured in milliseconds on the 2 test platforms (ARM A9 CPU, FPGA accelerator IP). The Y axis showing the execution time is in logarithmic scale.

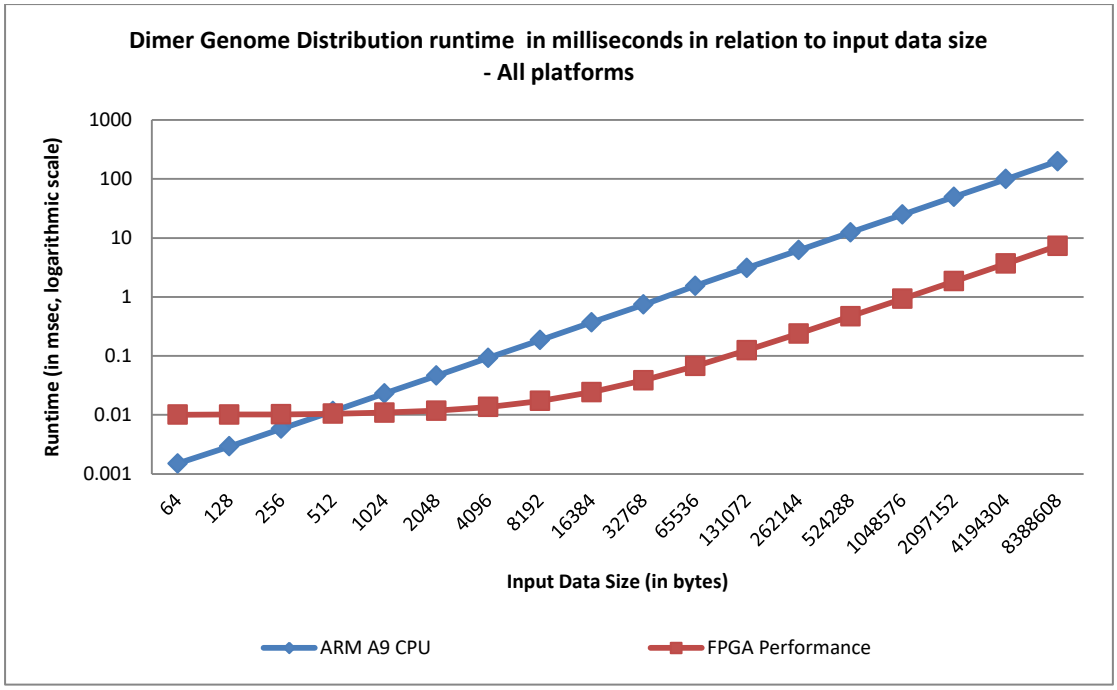


Figure 45 – Dimer genome distribution runtime, all platforms compared, semi-logarithmic graph

The FPGA implementation of the algorithm shows considerable speedup in the ranges of [16KB-8MB], with the FPGA implementation processing 8 MB of genome data (which translates to 8 million bases in the current encoding utilized) in 7.35 milliseconds while the ARMv7 implementation processes the same genome in 199.2 milliseconds, 27 times slower.

Similarly to the previous 2 algorithms, the FPGA implementation of the Dimer Genome Distribution shows a steady execution time of 10 microseconds for sizes of 64-2048 bytes due to the latency incurred from transferring data from the DDR to the PL.

The added 8 microseconds latency when compared to the latency of the BW Image Thresholding algorithm is attributed to the fact that there needs to be some preprocessing in the accelerator IP before the DMA engine starts sending actual genome data to the PL for processing. Additional latency is incurred due to post processing where we have to add all individual counters in the IP to a single array of dimer distribution counters before sending them to the DDR.

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- first_read_loop	21	21	3	-	-	7	no
- first_add_arr_loop	294	294	42	-	-	7	no
+ first_add_row_loop	40	40	10	-	-	4	no
++ first_add_col_loop	8	8	2	-	-	4	no
- first_init_arr_loop	294	294	42	-	-	7	no
+ first_init_row_loop	40	40	10	-	-	4	no
++ first_init_col_loop	8	8	2	-	-	4	no
- main_loop	32771	32771	5	1	1	32768	yes
- leftover_pairs_add_loop	32	32	4	-	-	8	no
- final_add_arr_loop	294	294	42	-	-	7	no
+ final_add_row_loop	40	40	10	-	-	4	no
++ final_add_col_loop	8	8	2	-	-	4	no
- send_loop_row	40	40	10	-	-	4	no
+ send_loop_col	8	8	2	-	-	4	no

Figure 46 – Timing information of Dimer Genome Distribution HLS IP

6.2. Performance throughput comparison

6.2.1. Black and White image Thresholding performance throughput

The graph below shows the performance in relation to input size of the data processed measured in MB/sec for the BW Image Thresholding application on the 2 test platforms.

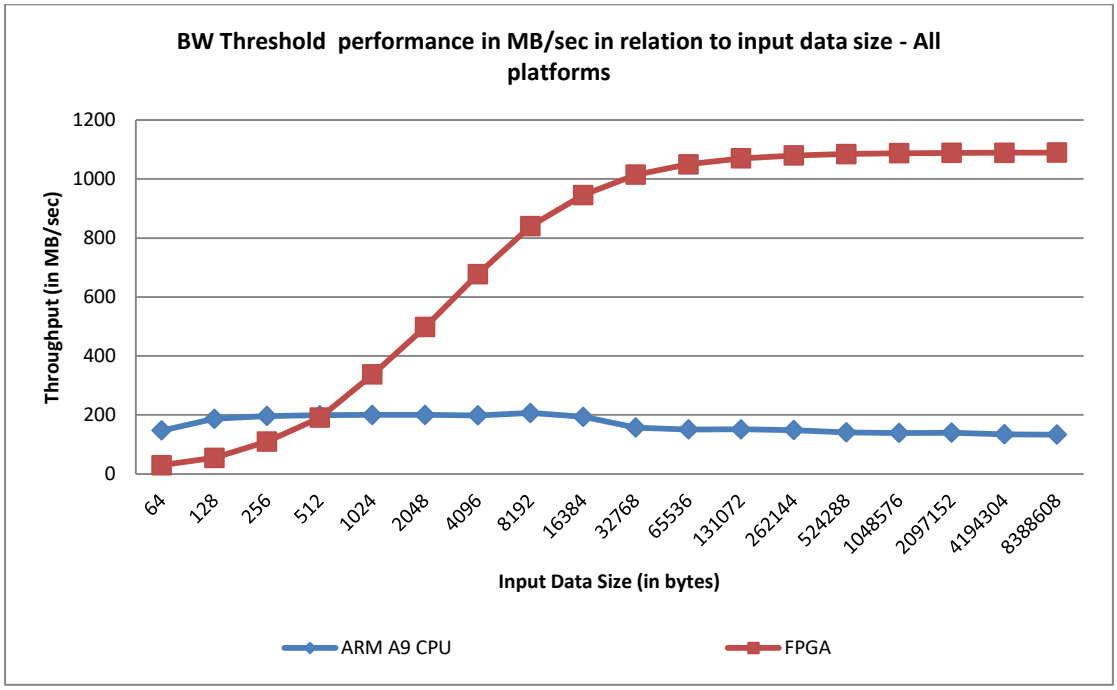


Figure 47 – BW threshold performance in MB/sec. All platforms. Sizes 64bytes-8MBs.

Throughput results show a clear performance benefit when utilizing the FPGA co-processor to calculate the BW threshold of an image. As the size of the input image increases, the FPGA throughput converges to the theoretical peak performance value of 1089.9135 MB/sec, with the 8MB input size computation reaching 99.997% of the theoretical maximum throughput.

6.2.2. Image Convolution with 3x3 kernel performance throughput

The graph below shows the performance in relation to input size of the data processed measured in MB/sec for the Image Convolution application on the 2 test platforms.

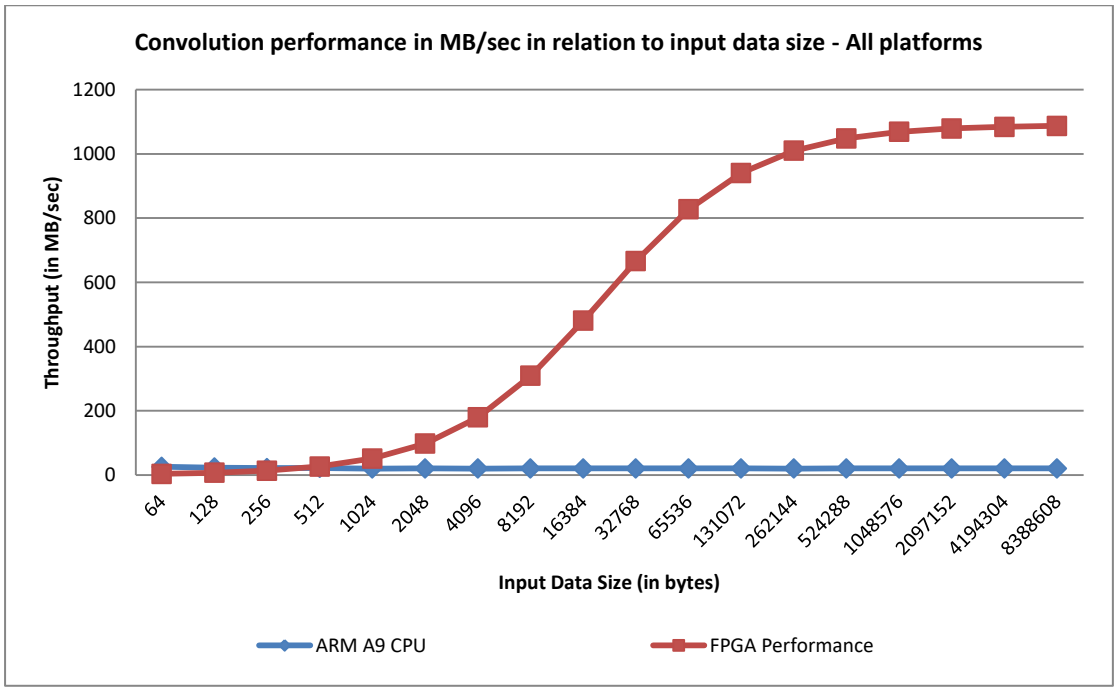


Figure 48 – Image Convolution performance in MB/sec. All platforms. Sizes 64bytes-8MBs.

Results of the image convolution application paint a different image in this case compared to the BW image Thresholding. Convolution requires many more operations applied on multiple data, some of them even reused during the process.

It is evident here that the FPGA with its concurrent computation capabilities and with the implementation of a sliding window buffer in the FPGA to allow resource reuse results in an implementation that is up to 5300% faster than the ARM implementation.

It is important to note however once again that for relatively small sizes of input data the FPGA performance drops dramatically. In the range of 64-256 bytes, the ARMv7 implementation is faster, although processing images that are so small may not be a common occurrence.

6.2.3. Dimer Genome Distribution performance throughput

The graph below shows the performance in relation to input size of the data processed measured in MB/sec for the Dimer Genome Distribution application on the 2 test platforms.

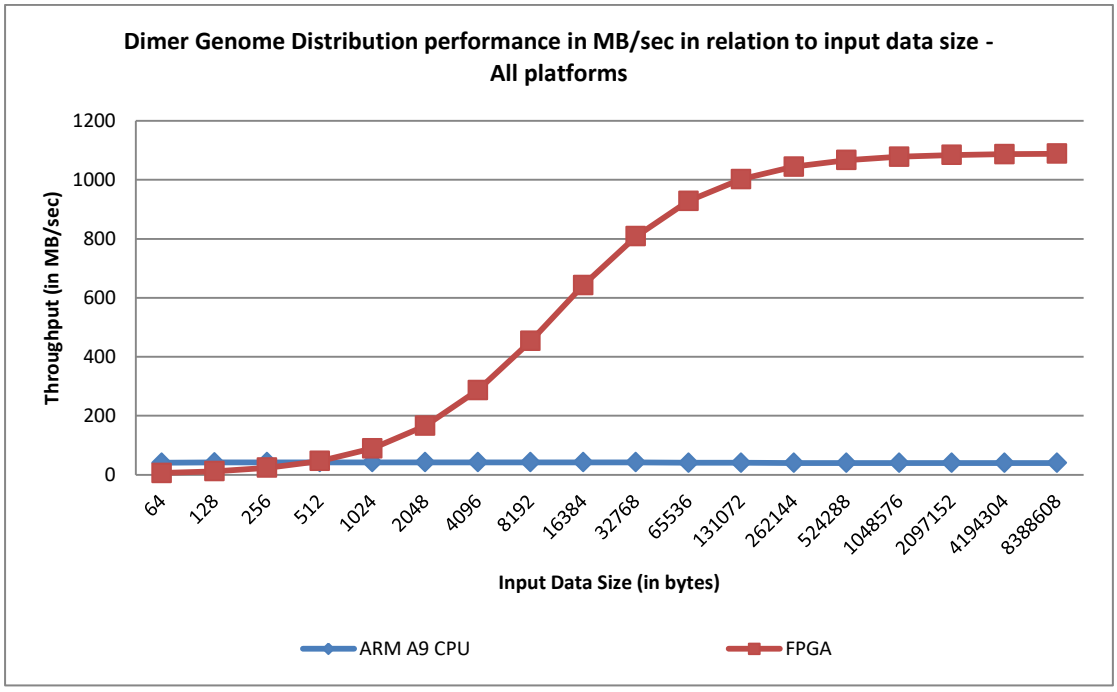


Figure 49 – Dimer Genome Distribution performance in MB/sec. All platforms. Sizes 64bytes-8MBs.

Similarly to the previous 2 algorithms, the FPGA implementation of the Dimer Genome Distribution algorithm is many times faster than the ARMv7 implementation.

The FPGA co-processor reaches 99.9% of the theoretical maximum when processing 8MB of genome data. This translates to a processing throughput of 8.38 Megabases/second (1 base = 1 byte). If the .2bit encoding was used we could theoretically achieve 4 times higher base processing throughput at the cost of increased resource utilization.

6.3. Energy efficiency comparison

6.3.1. Black and White image thresholding energy efficiency

The graph below shows the energy efficiency in relation to input size of the data processed measured in MB/joule (megabytes of input data processed per joule spent) for the BW Image Thresholding application on the 2 test platforms.

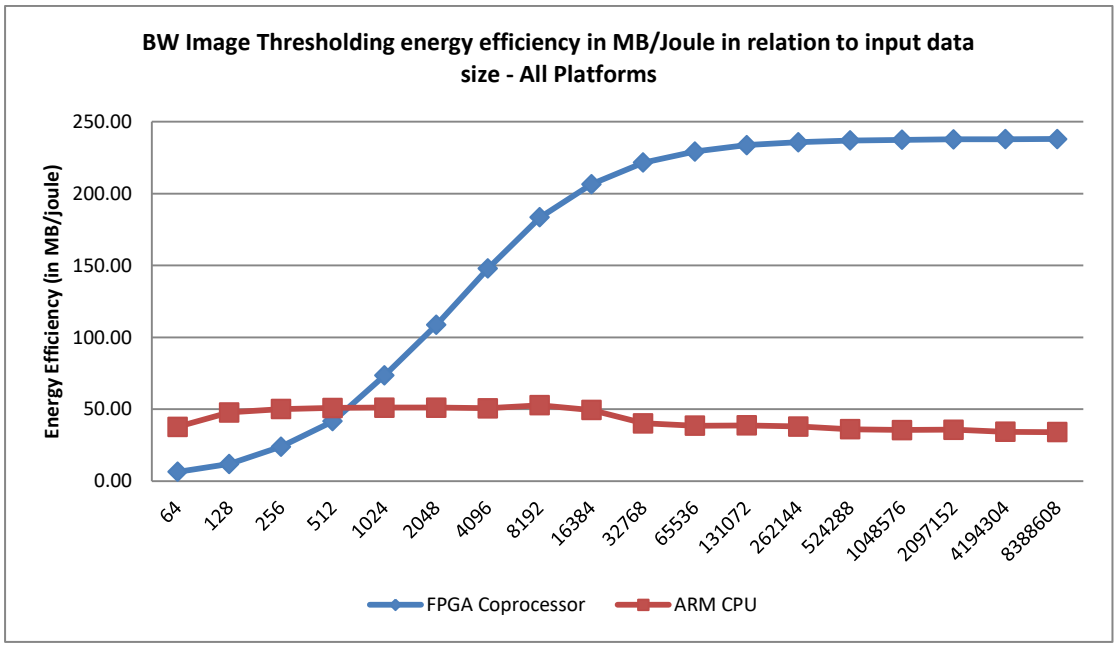


Figure 50 – Energy efficiency of BW Image Thresholding in relation to input data size on ARM CPU and FPGA HLS IP.

The FPGA implementation offers a clear advantage in power efficiency thanks to its much higher throughput. The device wattage is slightly higher when running on the FPGA coprocessor but because the runtime is several times faster, the overall energy expended is up to 7 times less than the energy expended to run the algorithm on the ARMv7 CPU.

Future FPGA implementations where the bit-width of the PS-PL port is higher such as Ultrascale+ devices or implementing a device hardware platform that can accommodate 200MHz of frequency in the PL could lead to dramatic increase in performance of the FPGA implementation.

6.3.2. Image Convolution with 3x3 kernel energy efficiency

The graph below shows the energy efficiency in relation to input size of the data processed measured in MB/joule (megabytes of input data processed per joule spent) for the Image Convolution application on the 2 test platforms.

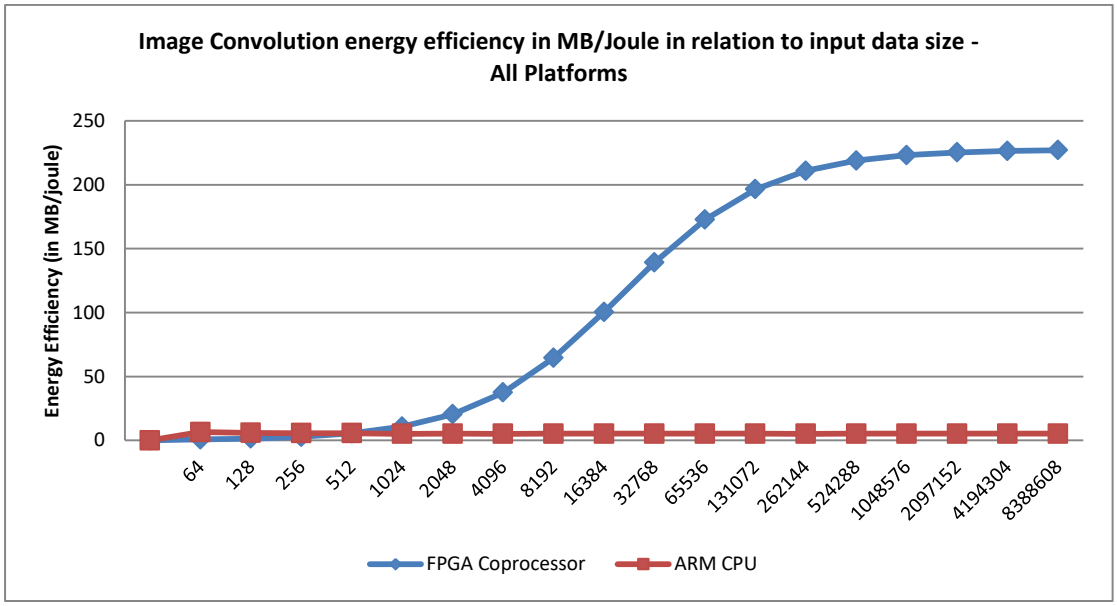


Figure 51 - Energy efficiency of Image Convolution in relation to input data size ARM CPU and FPGA HLS IP.

Similar to the BW Thresholding application, the FPGA is capable of outperforming the ARM processor in energy efficiency. However, this time the difference in efficiency is much more visible than the aforementioned application.

The FPGA implementation reaches 227 MB/joule energy efficiency when processing 8MB of data. Compared to the ARM CPU which has an energy efficiency of 5.22 MB/joule in the 8MB input size, the FPGA offers a x44 increase respectively in energy efficiency.

6.3.3. Dimer Genome Distribution energy efficiency

The graph below shows the energy efficiency in relation to input size of the data processed measured in MB/joule (megabytes of data processed per joule spent) for the Dimer Genome Distribution application on the 2 test platforms.

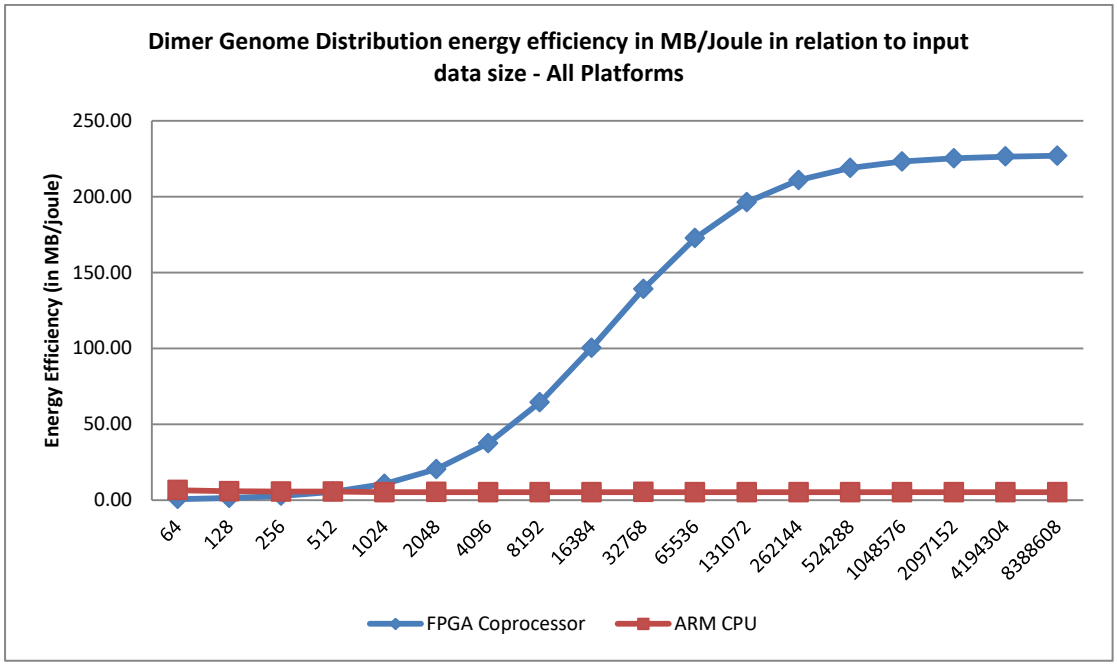


Figure 52 - Energy efficiency of Dimer Genome Distribution in relation to input data size ARM CPU and FPGA HLS IP.

Finally, Dimer Genome Distribution is similarly much more efficient when running on the FPGA thanks to a combination of both high parsing throughput of the genome as well as low power consumption.

Energy efficiency metrics are similar to the previous 2 algorithms, with the FPGA implementation offering up to 43x increase in energy efficiency when processing 8 MB of genome sequence data. Only for very small sequences is the ARMv7 implementation more energy efficient (ranges of 64-512 bytes of genome sequence data).

6.4. Cycles per byte performance comparison

6.4.1. Black and White image thresholding clock cycles/byte

The graph below shows the performance in clock cycles per byte processed for the BW Image Thresholding application on the 2 test platforms.

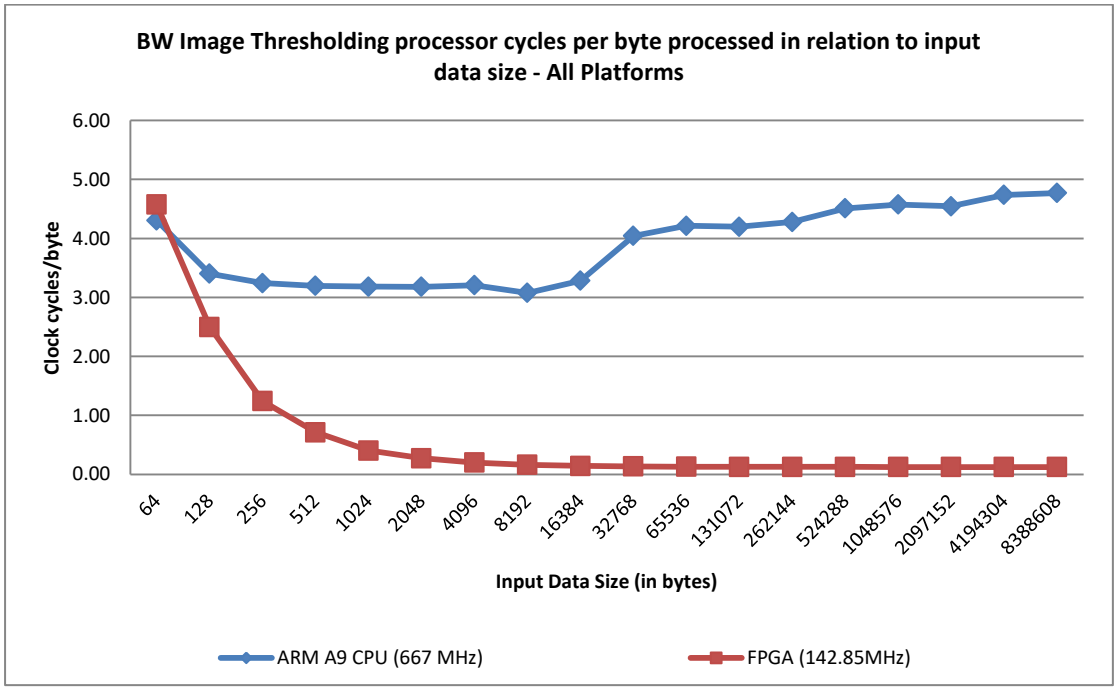


Figure 53 – Performance in cycles/byte of BW Image Thresholding in relation to input data size on ARM CPU and FPGA HLS IP.

Clock cycles per byte metrics show that the FPGA implementation offers unparalleled performance in concurrent execution of input data, managing up to a little over 0.125 cycles per byte processed, up to 38 times better performance than the implementation on the ARMv7.

The 0.125 cycles per byte performance of the FPGA indicates that an increase in clock frequency of the PL logic can result in a drastic increase in overall performance, as long as the bandwidth of the DDR module from which we read data and write output results can support it.

6.4.2. Image Convolution with 3x3 kernel clock cycles/byte performance

The graph below shows the performance in clock cycles per byte processed for the Image Convolution application on the 2 test platforms.

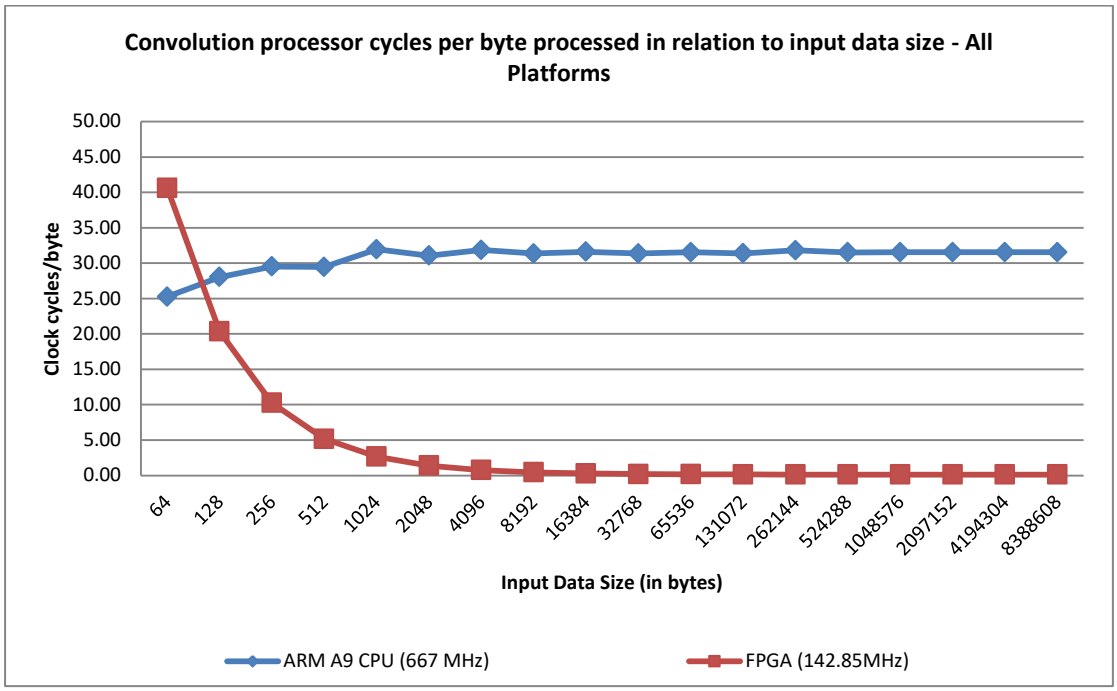


Figure 54 – Performance in cycles/byte of Image Convolution in relation to input data size on ARM CPU and FPGA HLS IP.

Image Convolution with 3x3 kernel implementation on the FPGA is much faster than the ARMv7 implementation even though the FPGA is clocked at a little less than 5 times lower clock frequency.

This means that the cycles/byte performance of the FPGA implementation can be up to 252 times better than the ARMv7 CPU implementation.

6.4.3. Dimer Genome Distribution clock cycles/byte performance

The graph below shows the performance in clock cycles per byte processed for the Dimer Genome Distribution application on the 2 test platforms.

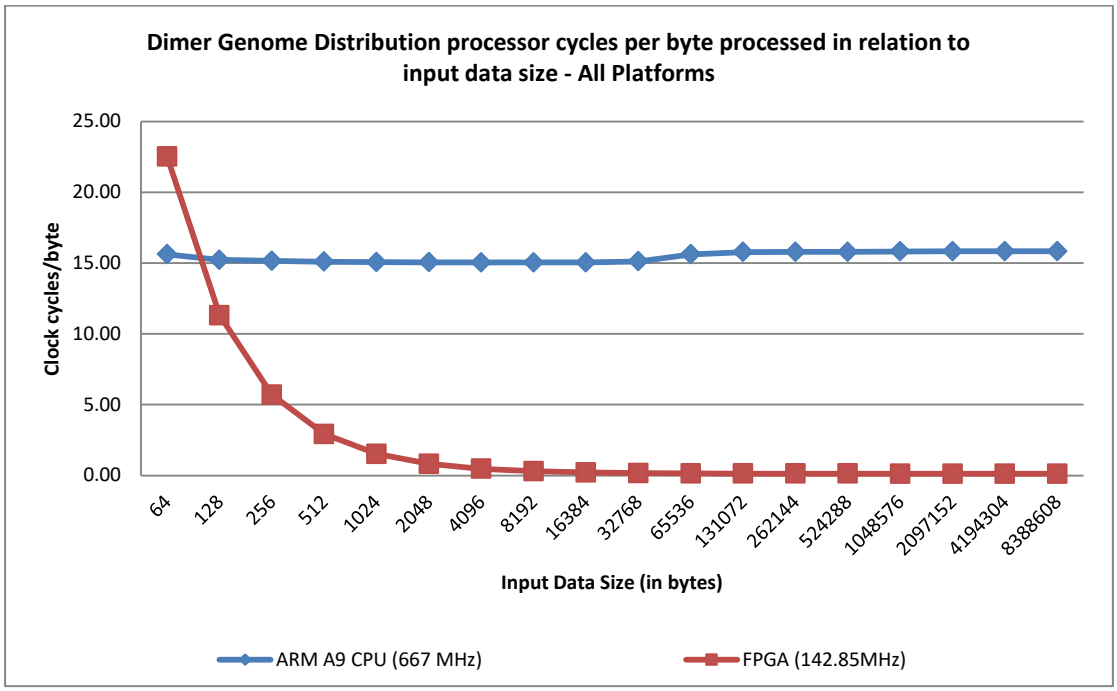


Figure 55 - Performance in cycles/byte of Dimer Genome Distribution in relation to input data size on ARM CPU and FPGA HLS IP.

Cycles/byte metrics for the 2-mer genome distribution application showcase that the FPGA implementation offers the lowest value of the 2 platforms. Similarly to the previous 2 algorithms for data sizes of 8KB and higher, it reaches its theoretical maximum of 0.125.

This shows promising results for implementations of the hardware platform that can run on 200MHz on the Zedboard or even for FPGA platforms that can be clocked at higher frequencies than the Zynq-7000 chip utilized in this work.

7. Conclusion and future work

7.1. Conclusion

Implementation results of the 3 algorithms that were executed on the 2 different platforms show measurements that coincide with results of past work.

As far as pure performance throughput metrics go, both in latency as well as average throughput in MB/sec of executing tasks of small size, it is clear that an ARM CPU is better than an FPGA. This is due to the fact that the data path from the DDR to the PL in an FPGA is longer and needs to pass through more interconnecting logic to reach the accelerator.

In their work on improvement of serving answers to web browser queries, Owaida et al. [57] showed that for input sizes of a few thousand or less scoring requests, the FPGA offers low bandwidth due to the static overhead of initiating transfers and kernel invocation time overhead. Because these metrics are mostly independent of input size however, as the input data size increases they take up smaller and smaller percentage of the overall time and the computation efficiency becomes higher.

However, for larger input data size, the FPGA can offer significantly better performance than the ARMv7 CPU across all 3 algorithms and in almost all measured benchmark metrics.

This of course can result in higher resource utilization on the FPGA; however the use of arbitrary precision structures of Vivado HLS when creating an IP can offer resource usage optimizations that will allow reconfigurable modules to fit in a defined Reconfigurable Region of a DPR-enabled design.

In regards to energy efficiency metrics, the expected results from past literature and reports indicate that the FPGA implementations of all 3 algorithms and for nearly all input data sizes offers much greater results than conventional architecture CPUs.

The most prevalent example of this can be seen when we compare the convolution of 8MBs of image data on the ARMv7 CPU and on the FPGA where the ARM CPU computes the convolution and consumes 1.53 Joules of energy while the FPGA implementation consumes 0.035 Joules, resulting in an overall increase in energy efficiency of 4271%.

ARMv7 implementation shows better results for very small input sizes, which are not expected to be common.

Clock cycles per byte processed metrics can indicate viability of increasing the clocking frequency of an FPGA device in order to increase performance throughput of

an implemented application or migrating the FPGA accelerator platform to a device that can handle higher frequencies.

The main contribution of this work is the implementation of a multi-disciplinary hardware acceleration platform on an FPGA that utilizes Dynamic Partial Reconfiguration and is designed to allow any type of Reconfigurable Module to be housed in a Reconfigurable Region.

Partial Reconfiguration constraints limit the reprogramming of a RP because the interface ports of each module need to be exactly the same in order for them to be compatible.

By transferring the metadata and the parameters of the computation process such as the threshold value in the Black and White Image Thresholding application or the Convolution Filter in the Image Convolution function through the same data port that the input data is transferred, we remove the obstacle of having to develop each acceleration function with the same interface ports.

Of course this means that the during development, the software handling delegation to the Programmable Logic coprocessors in later stages of the development cycle needs to be aware in which order each parameter is being sent.

To our knowledge, this is the first work that demonstrates this design paradigm to allow any function, regardless of the parameters that need to be passed in, to be included in a DPR-enabled FPGA hardware platform.

7.2. Future work

In this study we implemented a partial reconfiguration platform for offering computation acceleration services to users in a cloud computing environment. The use cases selected were much more energy efficient and cycle efficient on the FPGA platform than on the ARM CPU.

We believe that future work for the specific algorithms implemented should include the following

1. Black and White thresholding application should output data in a single bit per pixel format instead of an 8-bit value. This can help decrease write rates of the FPGA to the DDR and lower the consumed bandwidth of the DDR during processing of this application, freeing up bandwidth resources for other FPGA coprocessors.
2. The Dimer Genome Distribution application should be extended to allow processing of .2bit format genomes. This will allow the FPGA to showcase its bit-accurate processing capabilities and its viability as an acceleration platform for genome sequencing. Of course, this would result in more resources needed to be allocated for this coprocessor, meaning it may exceed resources of the

partial reconfigurations regions defined during floorplanning. This must be taken into account.

Besides the above, additional future work should focus on adding different, more computationally complex algorithms on the platform and evaluating their performance when compared to a CPU or GPU implementation. Theoretical maximum performance bandwidth given from Equation 4 can help designers evaluate the viability of migrating a task to the FPGA platform before beginning development on the hardware platform by first evaluating its performance on a conventional CPU or GPU platform.

Additionally, future work should focus on evaluating the performance of the scheduling algorithms used in this work to ensure resource reuse of the reconfigurable partitions and to select which partitions will be reprogrammed when it is needed. The fact that we use the Least Recently Used partition when a user requests an acceleration service may not offer the lowest overall probability that a reconfiguration will take place. It depends heavily on access patterns. For example, there may be cases where a Least Frequently Used (LFU) scheduling scheme offers better results.

References

[1] S. O'Sullivan, "Internet Solutions Division Strategy for Cloud Computing," 1996. [Online]. Available: https://s3.amazonaws.com/files.technologyreview.com/p/pub/legacy/compaq_cst_1996_0.pdf.

[2] X. Zenuni, J. Ajdari, F. Ismaili, and B. Raufi, "Cloud storage providers : A comparison review and evaluation Cloud Storage Providers : A Comparison Review and Evaluation," in *International Conference on Computer Systems and Technologies - CompSysTech'14 Cloud*, 2014, no. June, doi: 10.1145/2659532.2659609.

[3] L. M. Dang, J. Piran, D. Han, K. Min, and H. Moon, "A Survey on Internet of Things and Cloud Computing for Healthcare," *Electronics*, vol. 8, no. 7, pp. 1–49, 2019, doi: 10.3390/electronics8070768.

[4] G. Crespo-perez and A. Ojeda-castro, "Convergence Of Cloud Computing , Internet Of Things , And Machine Learning : The Future Of Decision Support Systems," *Int. J. Sci. Technol. Res.*, vol. 6, no. 7, 2017.

[5] J. Cito, P. Leitner, T. Fritz, and H. C. Gall, "The making of cloud applications: an empirical study on software development for the cloud," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, 2015, pp. 393–403, doi: 10.1145/2786805.2786826.

[6] P. Mvelase, H. Sithole, T. Modipa, and S. Mathaba, "The Economics of Cloud Computing : A Review," no. November, 2016, doi: 10.1109/ICACCE.2016.8073741.

[7] D. Lee, D. Kim, D. Kwon, and H. Kim, "Efficient Hardware Implementation of the Lightweight Block Encryption Algorithm LEA," *Sensors*, vol. 14, no. 1, pp. 975–994, 2014, doi: 10.3390/s140100975.

[8] C. Pal, A. Kotal, A. Samanta, A. Chakrabarti, and R. Ghosh, "An Efficient FPGA Implementation of Optimized Anisotropic Diffusion Filtering of Images," *Int. J. Reconfigurable Comput.*, vol. 2016, p. 17, 2016, doi: 10.1155/2016/3020473.

[9] Y. Said, T. Saidani, F. Smach, M. Atri, and H. Snoussi, "Embedded Real-Time Video Processing System on FPGA," 2012, doi: 10.1007/978-3-642-31254-0.

[10] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," 2011.

[11] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Futur. Gener. Comput. Syst.*, vol. 25, no. 6, pp. 599–616, 2009, doi: 10.1016/j.future.2008.12.001.

[12] Xilinx, "MicroBlaze Soft Processor Core." https://www.xilinx.com/products/design-tools/microblaze.html (accessed Jan. 03, 2020).

[13] C. Maxfield, *The Design Warrior's Guide to FPGAs*. Elsevier, 2004,

ISBN:9780750676045.

[14] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang, "Understanding Performance Differences of FPGAs and GPUs," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Apr. 2018, pp. 93–96, doi: 10.1109/FCCM.2018.00023.

[15] E. Nurvitadhi, D. Sheffield, Jaewoong Sim, A. Mishra, G. Venkatesh, and D. Marr, "Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC," in *2016 International Conference on Field-Programmable Technology (FPT)*, Dec. 2016, pp. 77–84, doi: 10.1109/FPT.2016.7929192.

[16] J. A. S. Laitner, "The Energy Efficiency Benefits and the Economic Imperative of ICT-Enabled Systems," 2015, pp. 37–48, http://link.springer.com/10.1007/978-3-319-09228-7_2.

[17] *Commission Regulation (EU) No 617/2013 of 26 June 2013 implementing Directive 2009/125/EC regading ecodesign requirements for computers and computer servers*. European Union Commision, 2013, p. 29.

[18] A. Andrae and T. Edler, "On Global Electricity Usage of Communication Technology: Trends to 2030," *Challenges*, vol. 6, no. 1, pp. 117–157, 2015, doi: 10.3390/challe6010117.

[19] M. Fahad, A. Shahid, R. R. Manumachu, and A. Lastovetsky, "A Comparative Study of Methods for Measurement of Energy of Computing," *Energies*, vol. 12, no. 11, p. 2204, Jun. 2019, doi: 10.3390/en12112204.

[20] P3International, "Kill-a-Watt Power Metering Device." http://www.p3international.com/products/p4400.html (accessed Aug. 21, 2020).

[21] Xilinx, "Xilinx Power Estimator," 2009. https://www.xilinx.com/products/technology/power/xpe.html (accessed Feb. 23, 2020).

[22] G. Kornaros (Editor), *Multi-Core Embedded Systems*. CRC Press/Taylor & Francis Group, 2010, ISBN:978-1-4398-1161-0.

[23] O. Vermesan *et al.*, "New Waves of IoT Technologies Research - Transcending Intelligence and Senses at the Edge to Create Multi Experience Environments," in *Internet of Things – The Call of the Edge - Everything Intelligent Everywhere*, DK: River Publishers, 2020, https://european-iot-pilots.eu/internet-of-things-the-call-of-the-edge-everything-intelligent-everywhere/.

[24] G. Kornaros and D. Pnevmatikatos, "A survey and taxonomy of on-chip monitoring of multicore systems-on-chip," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 18, no. 2, pp. 1–38, Mar. 2013, doi: 10.1145/2442087.2442088.

[25] M. D. Grammatikakis, G. Kornaros, and M. Coppola, "Power-Aware Multicore SoC and NoC Design," in *Multiprocessor System-on-Chip*, M. Hübner and J. Becker, Eds. New York, NY: Springer New York, 2011, pp. 167–193,

http://link.springer.com/10.1007/978-1-4419-6460-1.

[26] G. Kornaros and D. Pnevmatikatos, “Dynamic Power and Thermal Management of NoC-Based Heterogeneous MPSoCs,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 1, pp. 1–26, Feb. 2014, doi: 10.1145/2567658.

[27] G. Kornaros and D. Pnevmatikatos, “Hardware-assisted dynamic power and thermal management in multi-core SoCs,” in *Proceedings of the 21st edition of the great lakes symposium on Great lakes symposium on VLSI - GLSVLSI ’11*, 2011, p. 115, doi: 10.1145/1973009.1973033.

[28] I. Christoforakis, O. Tomoutzoglou, D. Bakoyiannis, and G. Kornaros, “Dithering-Based Power and Thermal Management on FPGA-Based Multi-core Embedded Systems,” in *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*, Oct. 2015, pp. 173–177, doi: 10.1109/EUC.2015.18.

[29] J. Yu, Y. Zhu, L. Xial, M. Qiu, Y. Ful, and G. Rongl, “Grounding High Efficiency Cloud Computing Architecture : HW-SW Co-Design and Implementation of a Stand-alone Web Server on FPGA,” in *Fourth International Conference on the Applications of Digital Information and Web Technologies*, 2011, pp. 124–129, doi: 10.1109/ICADIWT.2011.6041412.

[30] K. Eguro and R. Venkatesan, “FPGAs For Trusted Cloud Computing,” in *22nd International Conference on Field Programmable Logic and Applications*, 2012, pp. 63–70, doi: 10.1109/FPL.2012.6339242.

[31] F. Armknecht *et al.*, “A Guide to Fully Homomorphic Encryption,” *IACR Cryptol. ePrint Arch.*, vol. 2015, 2015.

[32] S. A. Fahmy, K. Vipin, and S. Shreejith, “Virtualized FPGA Accelerators for Efficient Cloud Computing,” in *International Conference on Cloud Computing Technology and Science*, 2015, pp. 430–435.

[33] A. Vaishnav, K. D. Pham, D. Koch, and J. Garside, “Resource Elastic Virtualization for FPGAs using OpenCL,” in *28th International Conference on Field Programmable Logic and Applications*, 2018, no. September, doi: 10.1109/FPL.2018.00028.

[34] A. W. Services, “Amazon EC2 F1.” Accessed: Feb. 22, 2020. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/.

[35] S. Karandikar *et al.*, “FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2018, pp. 29–42, doi: 10.1109/ISCA.2018.00014.

[36] Alibaba Cloud ECS, “Deep dive into alibaba cloud F3 FPGA as a service instances.,” 2018. .

[37] O. Tomoutzoglou, D. Mbakoyiannis, G. Kornaros, and M. Coppola, “Efficient Job Offloading in Heterogeneous Systems Through Hardware-Assisted Packet-

Based Dispatching and User-Level Runtime Infrastructure," *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 39, no. 5, pp. 1017–1030, May 2020, doi: 10.1109/TCAD.2019.2907912.

[38] D. Mbakoyiannis, O. Tomoutzoglou, and G. Kornaros, "Energy-Performance Considerations for Data Offloading to FPGA-Based Accelerators Over PCIe," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 1, pp. 1–24, Apr. 2018, doi: 10.1145/3180263.

[39] G. Kornaros and M. Coppola, "Enabling Efficient Job Dispatching in Accelerator-Extended Heterogeneous Systems with Unified Address Space," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Sep. 2018, pp. 180–188, doi: 10.1109/CAHPC.2018.8645945.

[40] G. Kornaros and M. Pratikakis, "VWQS: A dispatching mechanism of variable-size tasks in heterogeneous systems," in *2016 International Conference on High Performance Computing & Simulation (HPCS)*, Jul. 2016, pp. 196–203, doi: 10.1109/HPCSim.2016.7568335.

[41] Xilinx, "Vivado Design Suite User Guide: Partial Reconfiguration (UG909)," 2017, [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug909-vivado-partial-reconfiguration.pdf.

[42] M. Nguyen, R. Tamburo, S. Narasimhan, and J. C. Hoe, "Quantifying the Benefits of Dynamic Partial Reconfiguration for Embedded Vision Applications," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2019, pp. 129–135, doi: 10.1109/FPL.2019.00029.

[43] A. Nafkha and Y. Louet, "Accurate measurement of power consumption overhead during FPGA dynamic partial reconfiguration," in *2016 International Symposium on Wireless Communication Systems (ISWCS)*, Sep. 2016, vol. 2016-Octob, pp. 586–591, doi: 10.1109/ISWCS.2016.7600972.

[44] D. Luo, G. Pan, and G. Wang, "A Linux-based Dynamic Partial Reconfiguration System Applied on Xilinx Zynq," in *Proceedings of The 7th International Conference on Computer Engineering and Networks — PoS(CENet2017)*, Jul. 2017, no. July 2017, p. 047, doi: 10.22323/1.299.0047.

[45] Avnet, "Zedboard APSoC Integrated Circuit." https://www.xilinx.com/products/boards-and-kits/1-elhabt.html (accessed Jan. 02, 2020).

[46] Linux Foundation, "FPGA Manager Linux Kernel Documentation." https://www.kernel.org/doc/html/v4.18/driver-api/fpga/fpga-mgr.html (accessed Feb. 27, 2020).

[47] S. Neuendorffer, T. Li, and D. Wang, "Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Librarie," *Xilinx Wiki*, vol. 1167, p. 1, 2013, [Online]. Available: http://www.wiki.xilinx.com/XAPP1167.

[48] N.-M. Ho, E. Manogaran, W.-F. Wong, and A. Anoosheh, “Efficient floating point precision tuning for approximate computing,” in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2017, vol. 0, pp. 63–68, doi: 10.1109/ASPDAC.2017.7858297.

[49] N. Cristianini and M. Hahn, *Introduction to Computational Genomics*. Cambridge University Press, 2006, ISBN:9780521856034.

[50] NCBI, “RefSeq Genetic Sequence Database.” https://www.ncbi.nlm.nih.gov/refseq/ (accessed Feb. 28, 2020).

[51] F. Wu *et al.*, “A new coronavirus associated with human respiratory disease in China,” *Nature*, Feb. 2020, doi: 10.1038/s41586-020-2008-3.

[52] Xilinx, “Pipeline HLS Pragma,” 2017. https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/fde1504034360078.html (accessed Feb. 23, 2020).

[53] Xilinx, “Bootgen User Guide - UG1283,” 2018. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1283-bootgen-user-guide.pdf.

[54] B. Sean, “STB Image C Library,” 2014. https://github.com/nothings/stb (accessed Feb. 28, 2020).

[55] Adafruit, “INA219 Current Sensor.” https://www.adafruit.com/product/904 (accessed Oct. 18, 2020).

[56] Digilent, “ISNS20 Pmod Current Sensor.” https://store.digilentinc.com/pmod-isns20-20a-current-sensor/ (accessed Oct. 18, 2020).

[57] M. Owaida, G. Alonso, L. Fogliarini, A. Hock-Koon, and P.-E. Melet, “Lowering the latency of data processing pipelines through FPGA based hardware acceleration,” *Proc. VLDB Endow.*, vol. 13, no. 1, pp. 71–85, Sep. 2019, doi: 10.14778/3357377.3357383.