FPGA-Based Coprocessors for Clouds

JOSIP POPOVIC, B.Sc. EE

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs
in partial fulfillment of the requirements for the degree of

MASTER OF APPLIED SCIENCE
IN ELECTRICAL AND COMPUTER ENGINEERING

Ottawa-Carleton Institute for Electrical and Computer Engineering
Faculty of Engineering and Design
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada

Abstract

Computer Clouds are growing in size and customer base. Lately their growth has been limited or at least influenced by hosted server's power consumption.

FPGA is a good contender for providing task offloading to CPU based servers. However hard coded FPGA solutions are not flexible enough to support a variety of applications being used in Clouds. FPGA soft processors provide the required flexibility but not the speed. High Level Synthesis requires FPGA reconfiguration.

The solution to enabling use of FPGAs in Clouds seems to be logical: use a fundamentally simple and small processor (flexibility) that runs code that is as close to hardware fixed function architecture as possible (speed).

The above set of requirements has led to the creation of hardware architectures that improve the existing state-of-the-art with features applicable to the Cloud computing environment.

A real life design inspired by requirements from financial industry is implemented on a FPGA.

Table of Contents

Table of Figures

Table of Tables

List of Acronyms and Definitions

ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction set Processor
Cloud	An off-premise computing site. Compare to on-premise Data Center, a computing site within an organization.
CoP	Coprocessor
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit, main processor
CUDA	Compute Unified Device Architecture
DynaPath	Dynamically configurable Data Path. DynaPath offers interconnect services required for parallel processing. Electron uses DynaPath.
Electron	Processor architecture proposed in this thesis.
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPGPU	General Purpose application development for GPUs
HDL	Hardware Description Language
HLL	High Level Language
HLS	High Level Synthesis
HOL	Head of Line, often used as “HOL Blocking”
HPC	High Performance Computing
ILP	Instruction Level Parallelism
ISA	Instruction Set Architecture

Kernel	A highly parallel part of application to be offloaded to FPGA. In this thesis we refer to an Electron processor running nano-code as a Kernel.
LUT	An n-bit Look-Up Table. A hardware module used by FPGAs. It can encode any n-bit Boolean function as truth tables.
MOPS	Million of Operation per Second. In this thesis "operation" refers to one execution of Black-Scholes equation.
NUMA	Non-Uniform Memory Access
PBX	Processing Element Crossbar
PE	Processing Element
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
SIMD	Single Instruction Multiple Data
SOC	System on Chip
Soft IP	An Intellectual Property, a reusable 3rd party design that can be synthesised for any semiconductor process
TIE	Tensilica Instruction Extension
VHDL	VHISC Hardware Description Language
VHSIC	Very High Speed Integrated Chip
VLCW	Very Long Control Word
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration

List of FPGA Terms

The following is a list of important FPGA related terms used in this thesis.

Reconfigurable logic

Reconfigurable logic technologies such as FPGA allow hardware logic (gates) updates. At the same time hardware functionality is changed.

Reconfiguration

Reconfiguration is a process of changing FPGA logic. Reconfiguration requires a new FPGA configuration file. This new configuration file is created using FPGA synthesis and floor planning tools. While the new configuration file is downloaded to FPGA, FPGA becomes unusable and requires a reset after the configuration file is downloaded. The process of reconfiguration takes 100s of milliseconds. Creating the reconfiguration file may take hours.

Controllable hardware

Programmability is a word often associated with a processor running code; an example is a RISC processor running ISA instructions. In this project we use words “controllable”, “controllable <hardware | interconnect | architecture>”, “hardware controller” and “controllable hardware code” or just “control code”.

In the proposed architecture the controllability is a feature provided by FPGA hardware where a change of FPGA (BRAM) memory content creates different functionality mapped on the existing FPGA hardware via a controllable interconnect. This memory content is called “control code”. FPGA logic that performs data processing is called “controllable hardware” and it is made from a controllable interconnect (DynaPath) and controllable Processing Elements. A “hardware controller” or just “controller” is a scheduling state machine that executes control code and controls the hardware. For every new application

the control code is changed on the fly (FPGA reconfiguration not required). The controller is a fixed function hardware module.

Instead of “control code” term “nano code” is used. The word “nano” comes from the fact that this code is lower than micro code, where micro code is often used as the lowest level code in a hierarchy of programs. The nano code is running on the lowest level and controls hardware fixed functions.

Control code synthesis

The control code is created from a High Level Language (HLL) application code. HLL application code requires mapping to the existing FPGA controllable fixed functions. The process of mapping a HLL application code to existing FPGA hardware is called control code synthesis.

Controllable hardware templates

One FPGA instantiated controllable fixed function hardware or a set of available controllable hardware may not be sufficient, or may not have the most optimum set of features for every HLL application. Therefore a set of fixed function hardware is made available in the form of pre-synthesised templates (Virtual Hardware). If required an instantiated hardware component is replaced by a different template. The process of replacing templates is called the “partial reconfiguration”.

Chapter 1 - Introduction

1.1 Motivation

The motivation of this thesis is reduction of electrical energy consumed by the Cloud Computing installations.

Energy consumed by Clouds comes from two interconnecting sources: Cloud servers and the server room cooling. Minimizing server's power consumption would affect the cooling requirements. Consequently the Cloud reliability and financial bottom line would be positively improved. Environment would benefit from reduced carbon emissions from coal, a still dominant energy source.

In this thesis we focus on architectures and methodologies that would enable FPGA-based coprocessors to be added to Clouds. The architecture we arrived to is implemented on a development board and its benefits evaluated.

1.2 Cloud Power Consumption

Energy consumption in Clouds is growing [1]. It was indicated that two Google searches performed on a desktop PC generate enough CO_2 emissions to boil a cup of water [2]. Data Center energy consumption doubles every 5 years [3], while in 2005 the combined power and cooling required by Data Centers was 1% of the world's electricity [4]. Koomey reported that global data centers in 2010 accounted for 1.1% to 1.5% of total electricity use while in the US that number was between 1.7 and 2.2% [5]. Mills stated that the wireless Cloud, a system made from wireless devices and Clouds consume 10% of World's energy [1]. If Clouds were a country based on their energy consumption they would rate as the 5^{th} country, just in between Japan and India. A quote from this report "Data centers are the factories of the 21st century information age". Waste majority of this energy comes from coal - a non environment friendly and not unlimited source of energy.

High Performance Business and Technical Computing applications could be accelerated by FPGAs. These applications represent 20% of the server market. Power saved or speedup achieved with FPGA-based coprocessors versus running software application on servers varies. Many researchers reported that FPGAs consume fraction of the server power. 2-fold achieved speedup from FPGAs is not uncommon.

The following calculation cannot be taken as accurate; rather it is a guidance to give us some understanding of opportunities at hand.

$$WorldEnergySaved = totalDC \times appOffloaded \times (server - coprocessor)$$

$$WorldEnergySaved = 1.5\% \times 20\% \times (100\% - 10\%) = 0.27\%$$

Where:

WorldEnergySaved – a decrease in World Clouds and Data Center energy consumed if FPGA coprocessors are used instead of servers, *totalDC* – portion of the World energy consumed by Clouds and Data Centers, *appOffloaded* – percent of applications running in Clouds and Data Centers that are possible to offload, *server* – Energy consumed by servers. Here 100% is used since we are looking relative to coprocessor energy consumption. *coprocessor* - energy consumed by FPGA coprocessors as portion of energy consumed by servers. Here 10% is used. This number also includes the fact that coprocessors decrease the processing time.

From the above *WorldEnergySaved* equation, if FPGA coprocessors are used in Clouds/Data Centers the global world energy consumption would be decreased by 0.27%. Currently FPGAs are not widely used in Clouds. No significant proof of their presence was found. Yet opportunities appear to be huge.

Due to these high energy requirements, Clouds are being built near power plants. In some US states Clouds have a limited supply of energy which they cannot exceed. No effort is spared in reducing power requirements driven by the growth of users taking advantage of the convenience offered by Clouds.

One option to reduce power consumed by Cloud servers is to introduce hardware coprocessor blades. These machines would augment the server's CPUs with tasks where CPUs are not efficient. CPUs are built for flexibility; they can more or less do everything - therefore they lack hardware optimization techniques. Examples where CPUs could be offloaded and accelerated are found in the financial industry. Financial institutions often batch process massive amounts of data where many segments of complex equations could be processed in parallel or "for loops" unwrapped and executed on stream processing elements. These features are not available on CPUs.

FPGA-based coprocessors can perform these calculations in less time, consuming less power. Financial institution gains are twofold: more processing in less time and Cloud costs saving.

1.3 Reconfigurable Coprocessors in Clouds

Reconfigurable coprocessors have been researched and in use for some time. However they are still not widely deployed in Clouds [3]. An often quoted reason for this is lack of flow where SW programmers could take advantage of reconfigurable coprocessors without needing to take extra steps to learn how to use them.

There are two major motivating factors to add reconfigurable coprocessors to Clouds:

1. To do more in less time
2. To save electrical power

Two major enabling technologies are:

1. An efficient, FPGA-based, on-fly reprogrammable and reconfigurable platform enabling accelerated computation
2. A software centric flow where software designers can take advantage of reconfigurable platforms without knowing how they operate

Some of current attempts to address software centric flow include:

1. Soft multiprocessor cores residing on FPGAs. An embedded or cross-compiler creates code that runs on soft cores. This approach offers flexibility but not the processing throughput.
2. C-to-HDL tools. Application C code is transformed to Hardware Description Language (HDL). Standard FPGA flow is used to port HDL to FPGA hardware. This approach suffers from overhead time required to synthesize and configure FPGA.
3. Custom designed hardware with an interface to C-code. HDL development is time consuming, often measured in months.
4. Architectures with programmable control path and controllable and/or reconfigurable data path.

These and a few other methodologies are further analysed in Section 2.3.

1.4 Computationally intensive applications in Clouds

CPUs are not the best choice for applications requiring numerous calculations, such as Monte Carlo computations. Most generic server CPUs employ only one double precision floating point unit[1]. Random number generation is done is software – inherently slow. More complex equations get – CPUs need to do more work sequentially. Even multi-core CPUs cannot execute calculations well in parallel due to a requirement to exchange intermediate results among cores.

The solution to the above is in offloading the host CPU from tasks that require arithmetic processing of massive amount of data by exploiting parallel reconfigurable platforms such as FPGAs. FPGAs can be tailored to best match calculation requirements. They can also employ many arithmetic primitives required for parallel and pipelined processing as well as reconfigurable interconnects required for intermediate data exchange.

Application just being computation intensive may not be enough. They also need to have high computation-to-communication ratio [6]. FPGA coprocessors need processing data

[1] Some specialty processors such IBM Power7 is an exception to this rule providing 3 floating point units.

downloaded from the host system, time that represents the communication part of the computation-to-communication effectively reducing this ratio.

FPGAs lend themselves well to two main groups of CPU offloading tasks: Cloud maintenance specific and applications running in Clouds.

Cloud maintenance specific:

Wang et al. developed pvFPGA – a paravirtualized solution that directly virtualizes FPGA accelerators on x86 platforms [7]. The goal of pvFPGA is to speedup applications on the Cloud and it is built to work with Xen.

Virtual Machine (VM) migration - this is a process of moving VMs between physical servers. Purpose of the VM migration is Cloud system maintenance, load balance and server consolidation, fault tolerance and more [8].

The VM migration could be used for energy saving reasons. In this case VMs would be migrated to servers so some other servers would be freed up and could be placed into a sleep mode saving the Cloud energy. VM could be migrated to Clouds powered by renewable energy or to Clouds that require less cooling.

Problems with VM migration are: time required (service disruption), data security and energy consumed [9]. If VM are migrated over WAN networks (slower network speed) amount of VM data being migrated directly correlates with energy consumed. FPGA coprocessors could be used to improve all of the above VM migration issues. These coprocessors could run compression [10] and security algorithms [8] yet they consume less energy than servers. FPGAs fit well into streaming data processing. These FPGA features are likely to be used with compression and security algorithms.

Section 1.2 indicates that the following and similar applications take 20% of the Cloud throughput:

- Financial industry based modeling and High Frequency Trading [11][4][12][13][14][15][16][17][18],
- Real time process modeling (human physiology or chemical reactions [19]),

- Scientific computations ([20][21][22])
- CAD (SPICE analog electronic circuit simulation tool, ASIC design and simulation tools [23] [24] [25]) or
- Random process driven modeling (Monte Carlo used in financial simulations, SPICE [23][26])

1.5 Software Developers

Financial and scientific institutions have pools of highly skilled software developers specialized in their respective domains. They write proprietary code without the required knowledge or necessary development time allocated to deal with parallel hardware specifics - for that they rely on compilers, operating systems and underlying hardware platforms. Taking advantage of parallel hardware is well outside of their expertise.

There are two main approaches that could bridge software development expertise and reconfigurable hardware:

1. Use FPGAs as a platform to instantiate a number of soft processor cores (some examples are Altera NIOS, Tensilica Xtensa extensible cores) or
2. Use tools that translate C to Verilog/VHDL code suitable for FPGAs

Using soft processor cores as building blocks of a multi-core FPGA is currently a preferred solution, though inherently very slow. However, the same problem as with multi-core CPUs still exists: how to make sections of equations execute in parallel, how to exchange intermediate results and how to synchronize individual processors. Xtensa processors offer more flexibility versus traditional FPGA soft processors but they come with royalty fees, expertise in Tensilica TIE language is required, resulting processor need re-synthesis.

Use of C to gate or C to HDL tools (Forte, Catapult, C2S, Synphony, ImpulseC etc) is limited to `for`-loop algorithms, such as FIR [27] and do not do well with complex calculations. Complex calculations require state machine or light-weight processor support to schedule individual operations. Besides, to be used these tools require significant modifications to the existing application code [28].

We see the following two approaches as alternatives to the above:

1. Allow software designers to keep using standard C language where they would use pragma-operators to indicate segments of code that need to be offloaded. Dedicated pre-compiler-like tools would pre-process C code and map sections of it to hardware coprocessors. A requirement from software developers would be to analyze code and look for calculations that appear frequently and take most of the execution time [29]. A similar programming model is provided by OpenMP and by Convey Computer HC-1, an FPGA-based coprocessor. A step forward would be the development of a pre-compiler tool that would benchmark value of offloading parts of a code versus executing the same on the CPU.
2. Use of high level languages (HLL) that are meant to be used for parallel hardware, such as C or OpenCL. C and OpenCL libraries would be augmented with specially designed functions taking advantage of hardware coprocessors. Software developers need to have some awareness of hardware and write C or OpenCL programs accordingly.

Section 2.4.3 provides more information on this HLL to FPGA code methodology.

1.6 Objectives

Use of FPGAs in Clouds is a new research area. FPGAs consume a fraction of the power consumed by CPUs. However power savings alone may not be a sufficiently strong argument for Cloud providers to invest in FPGA-based machines. FPGAs can provide comparable or faster processing times than CPUs for data-parallel applications.

The thesis objectives are:

- Define requirements that would allow use of FPGAs in Clouds
- Suggest FPGA design methodologies that could be used in Clouds
- Define FPGA hardware architectures:
    - capable of accelerating applications beyond what is offered by CPUs,
    - flexible therefore they can run a significant portion of applications running in Clouds

    - provide low processing jitter in comparison to CPU based servers
- Show experimentally that the selected architecture provides application acceleration and lower power consumption versus a CPU
- Suggest FPGA level and system level techniques to improve computation throughput

A special focus is given, but not limited to, financial applications.

1.7 Overview of the Contributions

FPGA-based hardware architecture with programmable control path, controllable and optionally partially reconfigurable data path is the main topic of this thesis. This architecture provides required flexibility and run time acceleration for a variety of applications running in Cloud computing environment

The following contributions are made:

- An in-depth analysis of the existing FPGA design methodologies and their applicability to Cloud Computing was conducted.
- A set of Cloud coprocessor hardware architecture requirements were established.
- A Very Long Control Word (VLCW) processor named Electron is developed based on Cloud requirements. Electron is a low level processor that architecturally extends work done by the research community with new features applicable to Cloud computing: use of low level VLCW control word, it collapses 5-7 RISC processor pipeline stages to only 3 stages and supports multi-core processing. Multiple Electron instantiations form a multi-core structure called a Processing Tile.
- A unique use of a sparse output buffered crossbar in the Electron data path. The crossbar provides operand buffering effectively addressing a problem of variable processing elements latencies. The crossbar also allows execution of multiple parallel operations. The crossbar itself, crossbar control signals, registers and Processing Elements attached to it form a dynamically configurable data path – DynaPath.
- FPGA resource usage is analysed and techniques for balancing processing throughput and logic resources analysed.

- Electron has been implemented on a Xilinx development board. Practical implementation includes a multiprocessor system with 24 Electron processors. The board is connected to the PC using a PCIe bus.
- A number of experimental runs were performed comparing FPGA and CPU computation throughput and power consumption. FPGA throughput includes all communication overhead such as PCIe transfers.
- A system level throughput analysis revealed the development board limitations and from it a set of processing throughput improvements is defined.

1.8 Summary of Results

In the practical part of this thesis (see Chapter 5) two implementations of Black-Scholes equation are created, a multi-threaded software implementation running on the host CPU and a hardware implementation running on the FPGA development board. The FPGA implementation is based on an architecture developed in this thesis. Results obtained by the two implementations are compared using two metrics:

- Acceleration: execution speed expressed in Million Operations per Second [MOPS] and
- Power: absolute power consumed, Processing per Watt and Processing per Watt per Dollar

In this thesis an operation refers to one Black-Scholes equation computation. If some of most obvious ML605 limitations are removed results show that a Virtex 6 based ML605 development board has the throughput slightly larger than a contemporary 4-core Intel i5-3570 CPU running a multi-threaded software solution (turbo boost mode turned On). At the same time the FPGA offers 9.6 times better Performance per Watt.

1.9 Thesis Outline

Chapter 2 - Background gives a review of industry and academic FPGA design methodologies followed by a list of architecture requirements for Cloud. Based on these requirements an analysis of relevant FPGA architectural components is given. The Black-Scholes equation is presented as it is used in the practical part of this thesis.

Chapter 3 – Requirements, Thesis Scope and Contributions provides benefits and disadvantages of a number FPGA design methodologies, coprocessor requirements for Cloud environment, thesis scope and a list of thesis contributions to the current state-of-the-art.

Chapter 4 – FPGA-Based Coprocessor Architecture describes a proposed architecture that we expect to solve most of the issues and obstacles in successful deployment of FPGA-based coprocessors in Clouds. An important section in this chapter is the "Achievable System Throughput". In it we show how experimental results obtained on a development board presented here can be significantly improved.

Chapter 5 – FPGA Based Coprocessor Implementation presents all implementation details.

Chapter 6 – Experimental, Achievable and Theoretical Performance summarises all obtained results and compares CPU and FPGA-based solutions

Chapter 7 – Conclusions and Future Work gives ideas to improve and bring automation into the proposed flow

Chapter 2 - Background

In this section we present information required to understand sections following this one. The following topics are presented:

What is Cloud Computing? - Basic facts about Cloud Computing are provided.

Cloud Acceleration Technologies – CPU/GPU/FPGA technologie facts.

FPGA Design Methodologies - A review of the current FPGA design methodologies, followed by the methodology comparative analysis.

FPGA-Based Coprocessor Elements – This section provides an analysis of components required by the selected coprocessor hardware architecture.

Server Operating System Jitter – Here we describe one of objectives of this project and why it is important for Clouds.

Power Consumption in Integrated Circuits – In this section we provide information required to estimate power consumption increase due to the clock rate increase.

Amdahl's Law – Relevant information for this important guidance is given.

Implemented Application – Here we give all details relevant to application developed in the practical part of the thesis.

i5-3570 CPU – This CPU is used for CPU-to-FPGA comparison.

Development Board – Gives an overview of the ML605 development board we use in the practical part of this thesis.

PCIe core and Software Driver – This section describes open core IPs we use in this thesis.

2.1 What is Cloud Computing?

Cloud computing providers offer computing hardware and software as a service delivered over a network [30]. The three main services offered by Cloud computing are application (Google Docs, Microsoft Office 360), storage (Dropbox, S3) and connectivity (Webex, Yahoo email) [31]. However there are many other types of services that belong to the three main ones such as Software as a service (SAAS), Storage as a service (STaaS) or Desktop as a service (DssS). Examples of Cloud computing are [32]:

- Email on the go: Microsoft, Yahoo, Google etc.
- Data storage: DropBox, Humyo, Microsoft's SkyDrive, S3 from Amazon etc.
- Collaboration tools: Webex, Google Docs, Mikogo, Vyew etc.
- Virtual office: Google's online suite, Ajax13, ThinkFree, Microsoft's Office 365 etc.
- Extra processing power on demand: Amazon's EC2, Elastichosts, Rackspace Cloud, NASA's Nebula platform etc.

The most common quoted reason by businesses for using Clouds is to shift from IT capital[2] to operational expenses[3]. 75% of 572 businesses surveyed have adopted Cloud, while 90% expect to do so in the next few years [33].

Naturally, it is of paramount importance to Cloud computing providers to cut into operational expenses such as electricity and computing resources. Both of these can be addressed by using heterogeneous computing platforms that offer low power alternative or more processing power versus CPUs. The heterogeneous computing platforms allow more applications to be serviced in any given time thereby directly affecting the Clouds' ability to meet peak demands.

[2] Software licenses, servers and networking equipment
[3] Corporations pay for what they use

2.2 Cloud Acceleration Technologies

Clouds are populated with CPU based servers. In recent years hybrid systems consisting of CPUs and GPUs are gaining popularity. CPUs have a small number of powerful processing cores while GPUs are opposite having large number of smaller cores. CPUs are running serial program sections and GPU are running parallel program sections. These systems have found their use in general purpose scientific and engineering applications. Besides the accelerations they offer, GPUs popularity comes from the tool standardization and software design methodology that are similar to the standard C language development tools and programming model.

FPGAs with their available pipelining, parallelism and memory capacity can compete with GPUs while offering significant power consumption reduction [34]. The big question is how to make FPGAs more user-friendly for non Hardware Description Language (HDL) experts therefore they could use FPGAs without learning computer architectures or HDLs. The preferred way would be to have CPU/FPGA task partitioning and "hardware creation" done within the single programming environment [35] – similar to CPU/GPU task partitioning model.

Hardware coprocessors could be based on Graphical Processors (GPU) or FPGAs. FPGAs are comparable in speed to GPUs but consume less energy. Kestur et al. measured 2.7 to 293 times better energy efficiency on large data sets versus a CUDA GPU based platform [34]. It should be noted that they measured much better speed on CUDA for large data sets. However it appears that this is due to the application, complete data set is first downloaded to the FPGA before it is being processed[4] (latency hiding model was not used). Chey et al. analyzed and compared GPU and FPGA-based solutions [11]. This paper turned out to be very influential and inspired JP Morgan to partner with, and later to buy a 20% stake of Maxeler, an HPC (High Performance Computing) company. Maxeler has since developed custom FPGA solutions for JP Morgan [36]. Chen at al. created a real life case when

[4] Although this architecture is much easier to implement, it certainly greatly and adversely affects the system performance

$330/year could be saved per server if an FPGA coprocessor was used [37]. High Performance Business and Technical Computing (financial modelling, scientific computation and data analytics) applications could be offloaded and accelerated by FPGAs, representing 20% of the server market. Betkaoui et al. compared GPU with Convey Computer HC1, an FPGA-based coprocessor [38]. They concluded that GPU performed better with floating-point tests while HC-1 gave better results for non-sequential memory access tests. HC-1 provided superior performance and energy efficiency for embarrassingly parallel applications such as Monte Carlo Asian option pricing. Skalicky et al. compared CPU, GPU and FPGA performance [39]. In the conclusion it was indicated that all three technologies have distinctive area where they dominate.

Based on the above and other papers it appears that CPU, FPGA and GPUs could coexist [40]. Each technology has preferred application domains where they dominate over other technologies. However FPGA provides much better power savings over GPU and CPU.

Similar comparisons are made between FPGA hardware solutions (fixed function design) and FPGA embedded soft core based design [2]. Results demonstrated that the fixed function design consumed 300% less power than Altera NIOS based solution.

In order to save power in Clouds major server manufacturers plan to use many core chips based on low power Arm cores. Software compilers, application programs and operating systems need updates to take advantage of these chips. Arm based processors may not necessarily bring major speedup. Even if calculations that require accelerations can be divided and mapped to individual cores, there is still a problem of synchronizing cores efficiently [6].

The following are some of the major disadvantages of using FPGAs in Clouds:

1. Time required for downloading raw and uploading processed data from/to the host system. This time is not required if processing is done on the host system.
2. Required hardware expertise to use them. FPGA development expertise is very specific and often not available by the application software development houses.

3. Time required by the FPGA design flow methodologies that require synthesis, floor-planning, place and route followed by a device reconfiguration

These disadvantages need to be addressed by a chosen architecture. Section 3.1.2 considers the above disadvantages and proposes mechanisms to avoid them.

2.3 FPGA Design Methodologies

In this section some of the main FPGA design methodologies are presented:

- Custom (an application dedicated) design,
- Embedded processing cores,
- Design that supports pre-synthesised hardware templates (Virtual Hardware),
- High Level Synthesis and
- Academic solutions.

All of the documented FPGA design methodologies are analysed and their advantages and disadvantages identified as it pertains to the FPGA usage in Clouds. As a result of this analysis in the Section 3.1.2 we propose a set of features that FPGA architecture should support.

2.3.1 Custom design

Custom designs are designs dedicated to a particular application such as a DSP algorithm. Custom designs approach taking full advantage of the main FPGA features s pipelining and parallelism by defining the design's building blocks for exact purpose, such as optimized multiply and add modules, and often further optimizing logic to minimize resource usage or to support high clock rates.

Stojilovic et al. suggest a domain specific architecture [41]. It consists of a carefully crafted data path with processing elements (computation components) connected in a deep pipeline. Each pipeline stage has a number of processing elements. Selection of the processing elements in each stage is done statically for a given domain of applications in mind. This approach certainly is a step in correct direction since it offers some but not quite sufficient flexibility required for the multitude of applications running on Clouds. This

approach could be extended as a generic design with commonly used processing elements strategically selected in each pipeline stage. Due to the variety of applications running on Cloud this architecture would have one or more of the following issues:

- Still not have enough flexibility,
- Complex scheduling state machines,
- Low compute density due to poorly utilized processing elements.

Custom designs require great level of expertise in computer architectures and HDL programming while the development time is very long [20]. Fact is that custom design lack flexibility we are interested in for the Cloud environment; therefore we deem the custom design not to be applicable by its the definition.

2.3.2 Multiprocessing with embedded cores

An obvious way of bridging the gap between high level languages and FPGA is use of soft processor cores. A number of design houses offer soft processor cores and development tools:

- Arm Cortex – this technology is available on Xilinx and Altera FPGAs
- Altera - NIOS
- LEON – European Space Research and Technology Centre (can be used without licence fees)
- OpenSPARC T1 – Sun Microsystem, open-source
- Tensilica – offers configurable and extendable cores
- Xilinx – MicroBlaze

Complex computing machines developed in [42] and [19] are based on a high number of light processing cores (many-core, 30 to 400 resp.).

Processing cores can be interconnected with a crossbar [42]. Cores have private, local and global memory therefore exchange of data among cores is possible.

Individual processing cores do not need to be efficiently coupled if the device is geared towards parallel and decoupled computations required in modeling applications such as

human physiology or chemical reactions [19]. Simple processing cores based on NISC were developed and produced much better results versus MicroBlaze based application [19].

Results available from above and other papers seem to indicate that many-core based FPGA machines offer better power with similar features versus GPU or CPU based machines. However the processor **architecture** plays the **major** role. Standard processor cores such as NIOS or MicroBlaze do not fit into this category due to their low speed.

More on disadvantages of RISC processors versus Electron can be found in Section Chapter 3 - Requirements, Thesis Scope and Contributions.

2.3.3 Virtual Hardware

The hardware template based design is a process where a pre-defined and pre-synthesised hardware block can be added to a FPGA via the partial device reconfiguration. Templates are stored outside of the FPGA, on an on-board flash device or even the host memory, effectively offering run-time reconfiguration by using Virtual Hardware model [43] [35].

Templates could be:

- Hardware architectures performing a particular algorithmic task, such as speed optimized fused multiply and add operation. There could be a number of these hardware architectures, each having different latency, multi-threading capabilities and are of different size. In this thesis we are considering algorithmic Processing Elements for Virtual Hardware.
- Application specific accelerators, such a complex search algorithm. These accelerators are fixed function designs in a form of a pipeline or a state machine.

Groza et al. presented a Hardware Operating System (HOS) methodology [43]. HOS is a run time partial reconfiguration environment capable of managing parallel hardware blocks versus sequentially executing software processes controlled by a CPU and an OS.

FPGA taking advantage of the Virtual Hardware need to support a predefined way of interfacing (signalling) that the templates and FPGA host logic support. This prerequisite is

rather possible to meet if templates are arithmetic processing elements connected to a crossbar - the crossbar offers a standard interface.

2.3.4 High Level Synthesis

In this section we analyse High Level Synthesis (HLS) and High Level Languages (HLL). They are tightly coupled. HLS tool providers may require a specific coding style or coding constructs – therefore they support specific HLLs.

2.3.4.1 High Level Languages and High Level Synthesis Tools

In this section we look into tools that translate C[5] language programs into RTL. This is a very dynamic and currently active landscape with a number of manufacturers attempting to design a tool flow providing features and performance comparable to custom HDL design. These C language variants are referred to as High Level Languages (HLL) while the design flow is part of High Level Synthesis (HLS) flow.

HLLs for reconfigurable logic attempt to provide experts from many scientific fields with a familiar development environment and programming language. Within this development environment they could control or create FPGA-based devices using familiar programming syntax and semantics. FPGA features and resources would be accessed by a high level of abstraction programming constructs [44]. If this knowledge bridging is successful:

1. We could eliminate a design step which requires domain field experts to provide their perfected algorithms to FPGA experts for HDL coding (important).
2. Existing domain specific code base could be reused and accelerated on FPGA platforms. It is highly desirable to be able to compile legacy code with a minimum or preferably no change. Many lines of well written code have already been used, tested and proven in practice.

What is expected from HLS tools is to:

[5] Beside C language, Fortran tools are available but not considered here

- Extract processes from the C source code suitable for independent and potentially parallel hardware threads. These threads could come in forms such as: pipelined data paths or state machines or a combination of the two (important).
- If processes can run in parallel, create a number of threads optimizing required speed of execution and FPGA resources. More threads employed, faster program execution (important).
- If threads are independent but not parallel (operation dependencies), create a pipelined (sequential) architecture, where independent threads feed each other (important).
- In all of the above cases, synchronize thread execution and enable processed data and state (intermediate results) to be interchanged among threads (important).
- Output synthesisable HDL (RTL).

In addition to the above major features, a successful HLS flow would:

- Besides being good at creating fixed function hardware (e.g. DSP) it is expected from these tools to:
    - Create the instruction controlled data paths [45] (important). Currently HLS tools can only create FSMs[6] that control DSP functions (referred to as the "data path scheduling" used for latency and throughput control).
    - Create designs that are comparably area efficient as custom designs [46]. To this point this is not the case.
- Impose a minimum set of rules on C code development. Some tools require the use of pragmas, special data types or they support a limited set of C language (important).
- It is highly desirable to be able to compile legacy code with a minimum or preferably no change.
- Provide specialized and optimized HDL libraries, for example single precision floating point library.

[6] Information verified for CatapultC (Mentor/Calypto) and Vivado (Xilinx/AutoESL)

- Maximise use of expensive operations via resource sharing among threads (important),
- Reduce bit-width to minimum required for a given task

The above is an extensive set of features, not all are currently supported by HLS tools. Section 2.3.4.2 provides criticism related to HLS. Above items marked with (important) are those we in this thesis see as critical. They are added to Section 3.1.2. Most of the marked items are implemented in the practical part of this thesis.

Some of the former and current HLL players are:

- ACL – Altera OpenCL
- CARTE by SRC Computers
- CATAPULT-C by Mentor Graphics
- Cynthesizer by Forte
- DIME-C by Nallatech
- HANDEL-C by Celoxica
- IMPULSE C by Impulse Accelerated Technologies
- ACS by Maxeler[7]
- System-C, by Open SystemC Initiative
- Xilinx Vivado

2.3.4.1.1 ACL - Altera OpenCL

ACL creates hardware from OpenCL kernels. It also creates the memory sub-system and provides interconnect infrastructure that connects the memory subsystem and PCIe to the custom kernel logic. ACL is a system generating tool [47]. It generates the host to FPGA communication system, internal/external memory communication as well as the application itself. Gate synthesis, Place & Route and Timing Analysis are all part of the same flow.

[7] Oscar Mencer, Maxeler CEO, is author of [48] and [4]. There is no direct evidence Maxeler is using ACS

CPU programs are sequential. FPGAs are massively parallel. What ACL offers is a programming model to represent CPU and FPGA (heterogeneous system). Specific to ACL, a complete application code rewrite is required but coding does not need to be cycle accurate, therefore making ACL much easier to use than HDL.

2.3.4.1.2 Carte

Carte is a tool set developed by SRC Computers [48]. Carte takes C or Fortran code, translates and maps it to Implicit or Explicit execution units. Implicit computing refers to code executed on a processor while Explicit refers to sections of HLL code that is hardware accelerated on reconfigurable logic devices such as FPGA or CPLD.

Code that is mapped to reconfigurable logic is run on logic specifically created for an application during compile time. This logic needs to be synthesised to a bitmap and this bitmap need to be downloaded to the reconfigurable device.

Carte tool set needs to be used with SRC Computers designed hardware.

2.3.4.1.3 Catapult-C

Catapult-C was developed by Mentor Graphics. In August 2011 Mentor Graphics sold the tool to Calypto Design Systems.

Catapult-C provides HLL/HLS environment the most frequently used in the industry. It is used for algorithm and control logic[8] development, produces RTL targeted for ASIC or FPGA platforms. It can synthesise ANSI C/C++ without code extensions[9]. Pointer, classes and operator overloading is supported.

Its GUI allows analysis of the hardware circuit being created, and supports cycle based, RTL-based and gate-level based simulations. In addition Catapult-C can be integrated with SystemC allowing formal verification between generated RTL and C++ programs. This formal verification flow is of high importance for ASIC development.

---

[8] Such as Finite State Machines
[9] Meaning legacy code could be synthesized but potentially not fully optimized

2.3.4.1.4 Cynthesizer

Cynthesizer is a HLS tool developed by Forte Design Systems [49]. It takes SystemC as input. Cynthesizer can span the complete design flow with one source code (ESL to RTL-to GDSII) therefore providing confidence in design early on.

Since Cynthesizer produces RTL it does not depend on any platform and can be used in ASIC or FPGA development.

2.3.4.1.5 DIME-C

DIME-C is an HDL tool developed and distributed by Nallatech. It can be used as a part of DIMETalk design flow or as a standalone C to VHDL compiler for Xilinx Virtex 4 and 5 FPGAs. If used as a part of DIMETalk libraries are provided for accessing Intel Front Side Bus.

Nallatech indicates that DIME-C should be used to design and optimize algorithms[10]. In addition the tool supports: subset of ANSI-C, pipelined and parallelised designs, algorithmic libraries as well as single and double precision floating point numbers. Pointer support is not provided. Legacy code may need to be modified before using DIME-C.

2.3.4.1.6 Handel-C

Handel-C is owned and sold by Mentor Graphics [50]. Originally it was created in 2008 by Celoxica. Celoxica was acquired by Agility who ceased to operate in 2009. In 2009 Mentor Graphics purchased Agility's C synthesis technology.

This tool flow is used to accelerate development, optimization and acceleration of DSP algorithms. The main tool feature is that the same HLL code can be compiled to a number of design languages (such as Verilog), synthesised and executed on a CPU or on embedded

[10] Algorithms such as DSP. This statement can be applied to almost all HLL tools. State machine support or arbitration appears to be better if done in HDL.

systems. Mentor Graphics provides embedded systems and libraries[11] optimized for Handle-C however the tool itself can be used for any FPGA platform.

2.3.4.1.7 Impulse C (Streams-C)

Impulse C is owned and sold by Impulse Accelerated Technologies [51]. Streams-C HLL tool developed at Los Alamos National Laboratory [52][53] was licensed to Impulse Accelerated Technologies under the name Impulse C.

Impulse C is a C-to-Gates tools that allows an HLL application to be accelerated by compiling it into a number of parallel processes that can be run on an FPGA, FPGAs or a combination of FPGAs, embedded microprocessors or DSP processors [54]. In order for an application to run on this mix of FPGA/processor systems Impulse-S provides features such as communications and process synchronization via shared memories, signals, message passing or streams. This model is referred to as a dataflow or as stream processing and takes advantage of the FIFO to accomplish the task of inter-process communication and process synchronization.

Impulse C comes with math and image processing libraries. The tool is not dedicated to any particular ASIC/FPGA-based products.

2.3.4.1.8 Maxeler

Maxeler is a commercial enterprise offering specialised HPC platforms [4][55][56]. These high end computers are built from FPGAs. FPGAs contain hardware processing elements connected in parallel and in a pipeline fashion. Therefore processing elements provide dataflow architecture allowing for streaming data processing. Maxeler uses the Ford assembly plant as a way to describe how dataflow processing takes place. Each of the assembly plant stations provides specific processing capabilities while data is flowing through the assembly plant.

[11] IO libraries for example are heavily dependent on FPGA platform. MG provides them for their RC Series platforms.

Maxeler toolset takes Java application code as input; Java code is compiled with Maxelers' proprietary MaxCompiler. The compiler outputs a configuration file that describes processing elements, and how they are interconnected. Intermediate results obtained by individual processing elements are forwarded to a downstream processing element rather than being written to an off chip memory.

Maxeler products are used in applications requiring computation acceleration such as: the Oil & Gas industry, financial analytics, low latency security trading and scientific computations.

JP Morgan Stanley, a financial industry giant, is a significant Maxeler shareholder. This is significant proof of the value HPC FPGA-based acceleration and HLL design flow has to offer. However the fact that Maxeler toolset works only with Maxeler hardware cannot be neglected.

2.3.4.1.9 System-C

SystemC is an open standard language used, among other applications, for HLS. It has similar semantics to VHDL/Verilog[12]. Main features are support for concurrent processes, C++ and user defined data types, clock cycle accuracy and 4-level logic (1, 0, X and Z). A number of silicon tool vendors (Cynthesizer) support SystemC as an HLL input for synthesis or for verification purposes. However it appears that its main use is in ESL due to its cycle and transaction level accuracy.

2.3.4.1.10 Vivado

Vivado is a design suite offered by Xilinx. It includes Vivado HLS (re-branded AutoESL), an HLS tool that translates a high level language (C or C++) to RTL. Berkeley Design Technology, Navarro et al. and Hiraiwa et al. use Vivado HLS exclusively in DSP applications [57][58][59]. The design goal is to replace a microprocessor and DSP processor with an FPGA while improving performance and lowering consumed energy. Vivado HLS produces

[12] This apparently is one of the reasons why SystemC never made it to HDL user community. VHDL/Verilog developers did not want to switch from HDL to SystemC since SystemC operated on the similar level of abstraction.

a significant improvement in processing speed. Berkeley Design Technology reported a forty fold performance increase in video DSP application versus a DSP processor. Hand coded HDL implementation will produce similar throughput while requiring 3 to 4 times fewer FPGA resources. However HDL implementation will take 3 to 4 times longer to develop. When an update (design exploration or a new standard) of a DSP algorithm is required, the combination of Vivado HLS and FPGA will offer the required flexibility.

Reusing the code developed for the CPU will not lead to a practical HLS design. Therefore manual reiterative restructuring of the CPU code is a mandatory step [57].

2.3.4.1.11 Academic HLS Tools

In this section a few academic solutions are presented. It is worth noting that a number of described commercial solutions, Section 2.3.4.1, were launched in research communities (Maxeler and Impulse C for example).

2.3.4.1.11.1 Open Reconfigurable Computer Language (OpenRCL)

OpenRCL developers claim that the OpenRCL compiler does not require modification to the original C code [42]. The compiler requires neither the use of pragmas in the application C program nor the use of hardware-like semantics that other academic/commercial tools require. However the toolset is geared towards a specific hardware consisting of parallel Processing Elements that are based on a standard ISA processor with reconfigurable datapath width and extensible ISA. Individual processing elements exchange data via a cross bar or globally accessible memory hierarchies.

The design flow includes:

- Main C/C++/Fortran GCC front end, LLVM and code generator and linker optimization of the main control code,
- OpenCL Kernel code GCC front end, Kernel memory access optimization, Kernel scheduling and code generator and linker

2.3.4.1.11.2 SA-C

Single Assignment C (SA-C) is an HLL compiler initiative by Colorado University [60]. SA-C is a compiler for a variant of C language that maps HLL programs onto FPGAs. SA-C

compiles C programs to an FPGA configuration file, generates the host code to download the configuration to FPGA, downloads the data and run-time parameters, starts the coprocessor and uploads results.

In addition, for more sophisticated FPGA users, SA-C can generate intermediate data flow graphs that can be viewed with tools provided.

2.3.4.1.11.3 Silicon to OpenCL (SOpenC)

Falcao et al. [23] and Owaida et al. [24] describe SOpenCL(Silicon to OpenCL) toolsets. The tool has two main processing steps. The first step converts OpenCL kernels to a manageable number of kernels so they could be mapped to an FPGA. This step is a source to source code translation.

The second step maps code from step one to an FPGA architecture template. A template is a top level HDL instantiations and contains interconnect structures such multiplexors and buffers. The exact template selected is based on C code program requirements. The template is instantiated on an FPGA. Next SOpenCL translates the C code to a synthesizable HDL (high level synthesis flow). What follows is a partial FPGA reconfiguration.

2.3.4.1.11.4 C-to-Verilog (C2Verilog)

C-to-Verilog is a free open source and on-line available C compiler [61]. C code needs to be copied and passed to the C2Verilog web site which will result in the creation of an optimized and synthesis Verilog code.

2.3.4.1.11.5 FPGA CUDA (FCUDA)

In this design approach a CUDA application is taken through a FCUDA flow producing synthesisable RTL code [62]. FCUDA takes advantage of fine and coarse grain parallelism available from the CUDA programming model. The flow has 4 main steps:

- CUDA code is source-to-source code translated to FCUDA annotated code.
- FCUDA annotated code is compiled by the FCUDA compiler preserving coarse-grained parallelism (thread-blocks level).
- AutoPilot C code synthesis tool is used to create RTL. AutoPilot extracts fine grain parallelism and ports it into created RTL.

- FPGA-based RTL to gate synthesis tool.

2.3.4.2 High Level Synthesis Criticism

Curreri et al. and Sanchez-Roman et al. reported their experience with Impulse C [63][64][65]. Sanchez-Roman et al. started with legacy C code developed for a 2D Navier-Stokes solver used in Computational Fluid Dynamics (CFD). FPGA platform is XtremeData XD2000i In-Socket Accelerator. This accelerator uses CPU Front Side Bus reducing the host system CPU to FPGA communication overhead. They achieved 22 times speedup versus software implementation and an order of magnitude energy savings[13].

They concluded that HLS does reduce development time versus HDLs (by a factor of 34 times) but C code manual optimizations are prerequisite. C code profiling was used to find bottlenecks and sections of C code to implement in FPGA. This step was followed by changing the C code:

- Data storage had to be manually changed for parallel hardware processing,
- Manual transformation of equations to extract common factors,
- Removing as many division operations as possible. For example if a common denominator was used, its inverse was calculated once only and the multiplication operation was used.

Although it is possible to understand that authors had to do manual changes relevant to data storage, it is surprising that Impulse C could not perform equation and division related transformations.

Another reported Impulse C issue; the tool would aggressively place pipelines to achieve one result per clock – although memory sub-system could not deliver data with the same speed (Section 4.8). At the same time selection of mathematical primitives was not adequate. For example, multiplying or dividing by two, was solved by instantiating regular multiplier or a divider.

[13] Provided speedup numbers do not include Host to FPGA communication time. This paper is not about CPU acceleration.

Lin et al claim that HLS tools (CatapultC, C2Verilog, Impulse C etc) still require significant hardware expertise, the expertise that is much different than standard C code development [42]. In [19] it was reported that the same application designed by an industry HLS tool would generate substantial logic in comparison to a custom design.

2.3.5 Academic Solutions

Next we document a number of academic solutions and ideas that could contribute to successful FPGA deployment in the Cloud.

2.3.5.1 No Instruction Set Computer (NISC)

If the NISC [66] processor was using an instruction set it would belong to the ASIP architecture family. NISC has a custom data path designed to meet application requirements, just as ASIPs do, however NISC does not have an instruction set. On the other hand, the NISC data path does have a number of processing elements versus a standard RISC architecture which only has one ALU.

Figure 1 shows NISC architecture [66]:

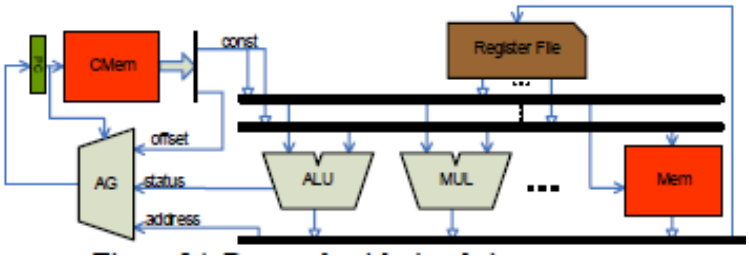

Figure 1 - NISC architecture [66]

NISC has separate control and data paths. The control path is constructed from CMEM (Control Memory), PC (Program Counter) and AG (Address Generator). The data path is constructed from RF (Register File), PE (Processing Elements such as ALU, Multipliers, Dividers and Comparators), Mem (local Memory) and hardwired interconnects.

In a traditional ISA processor it is the compiler's responsibility [67] to maximize the processors' data path utilization by performing code optimizations, such as register renaming or scheduling long latency instructions. In the NISC architecture, a designer is

given an opportunity to customize the data path while the NISC compiler generates and schedules clock accurate instructions.

In order to understand NISC instructions a standard ISA instruction format is shown in Figure 2. Consider the following example:

Rd(r24w) = Rs(r0w) + Rt(r8w)

Where: Rd – destination register, in this case register whose byte address is 24 and width is one word (4 bytes starting from 24 and ending on 27), Rs (made from registers 0, 1, 2 and 3) and Rt (registers 8, 9, 10 and 11) are two source registers.

This instruction would be represented in the processor's instruction memory as:

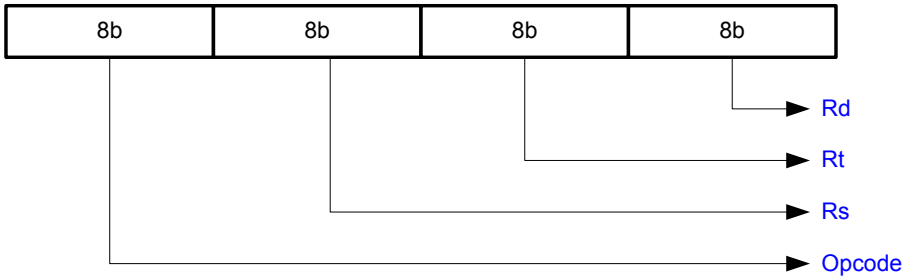

Figure 2 - ISA Instruction Format

where the Opcode (operational code) is an addition operation.

In the above case an ISA processor's decoder would decode the instructions' Opcode, create control signals and output them to the processor's execution pipe.

In NISC architecture the above instruction would be represented exactly like an ISA decoder stage would: as a set of control signals [68] demonstrating that the compiler directly controls data path components.

These control signals are connected to a customized processor's data path. Therefore the instruction format is unique for a given data path allowing room to clock speed and

throughput optimizations, while providing quick turnaround experimentation with different data paths.

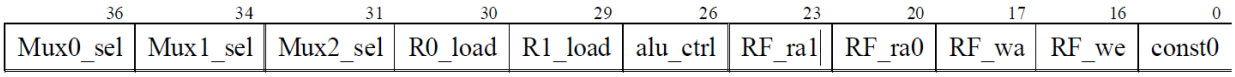

Figure 3 - NISC instruction word example [68]

From the above figure:

- MuxN_sel – a 2-bit control word configuring multiplexor N
- Rn _load– a 1-bit control signal used to latch an input to register N.
- RF_r/wN – a 3-bit control word that controls Register File port N. Note the Register File is a 4-port memory.
- const0 – a 16-bit constant

The NISC compiler reads the application C code and creates a data flow graph (DFG) from the C code. DFG operations are clock cycle accurate scheduled while DFG operators are bind to data path components. NISC application examples are provided in [69] [70] [71] [72] [73] [67]. The NISC processors' data path is optimized for a given application and benchmarked against RISC processors. Clock count, power and energy improvements are documented.

Papers [68] [74] [75] document NISC design flow, NISC toolset, automated data path trimming and other optimisation techniques.

It should be noted that NISC does not employ any type of handshaking between processing elements in order to signal if data is available for processing. NISC architecture relies on control and data path latencies. These numbers have to be constant and known a priori. If latencies change, control logic and control code need to be recompiled.

2.3.5.2 FlexCore

The FlexCore architecture is very similar to NISC [76]. It is based on a full size interconnect that connects all functional unit with each other or themselves. Functional units (Register

Array, Load/Store unit, ALU) are exposed to a controller. This controller runs code that controls all functional units and data flow among them.

FlexCore authors reported a 40% speedup over a RISC GPP (General Purpose Processor) for a number of embedded application domain benchmarks. They also note that their instruction memory is not very efficient in terms of size. Reason for this is that FlexCore uses a VLIW concept called Native-ISA (N-ISA). N-ISA is decoded instruction controlling all functional units.

2.3.5.3 Variable Processing Latencies

This chapter suggests solution to the problem of variable processing latencies. Variable processing latencies are a scheduling problem where computations completed in different parallel branches need to be used as operands and therefore clock aligned.

Hong et al. suggest architecture where Processing Elements (PEs) are interconnected by buffers [77]. Buffers provide processing latency synchronization between data producers and consumers. The complete system is controlled by a global counter based controller.

Hong et al. point to differing latencies among PEs (there are different numbers of pipelines or the PEs use different clock frequencies) [78]. There are 2 approaches to synchronize PEs: centralized (long wiring issue; small change in one PE creates churn all over the design) or distributed controllers[14]. If the DFG path allows freedom in re-arranging operations, path delays are equalized minimizing buffer size requirements. If PE latency allows, one could use PEs at a lower clock rate since this could allow for architecture choices that consume less power. Buffer controllers need to start writing or reading from buffers to minimize system latency, and to prevent overflow (by activating reads) and underflow (by activating writes).

Hong et al. also proposed that PEs could be processors or fixed function logic [79]. System level synchronization among PEs is achieved with buffers. Fixed function PE updates

[14] In the practical part of this thesis a combination of the two is used

require FPGA reconfiguration. The reconfiguration can be static or dynamic. Static configuration is loaded during the system initialization. Dynamic reconfiguration is defined as global or partial. Partial is done with distributed memories. Hardware needs to be efficiently partitioned to allow for meaningful partial reconfiguration.

Hong et al. [80] suggest a design approach to address issues with current DSP processors. DSP processors are highly pipelined and support some level of concurrency but they are still not suitable for algorithms that have different structure such as particle filtering that are computationally complex algorithms where integration is performed using Monte Carlo techniques. The goal is to design a programmable particle filtering logic[15] that is good for more than one particle filtering algorithm while outperforming implementation of these filters on DSP processors. In this suggested design approach PEs are interconnected with distributed buffers, controllers and multiplexers. Individual PEs can be switched in and out depending on a particle filtering algorithm.

Hong et al. [81] extracted processing functionality from an application data flow graph and mapped it to a number of processing blocks. Processing blocks are implemented as soft processor cores or fixed functions. The main challenge presented by processor cores is that due to different code paths, core processing time is not known a priori. Therefore a mechanism of exchanging processed data between processing cores and fixed function logic is required. Implementing buffers and buffer synchronization methods is examined as a solution to this problem. Buffers are controlled by local (distributed) controllers while the complete system is synchronized by a centralised controller. Processor cores interface with the global controller. The global controller schedules the local buffer controllers. The main purpose of this project was processing block scheduling and synchronization, not a reconfiguration. Soft cores should offer most of this functionality.

[15] Domain processor

2.3.5.4 Other relevant projects

FPGA logic is synthesised from an application data flow graph [82]. Individual Data Flow Graph (DFG) nodes are designed as processing elements of particular complexities and processing latencies. Processing elements are defined as one of: combinational logic, single-cycle sequential logic or multi-cycle sequential logic. Combinational logic and single-cycle sequential logic have execution times of 0 or one clock (respectively) while different multi-cycle sequential logic elements have fixed execution time of different values. In order to synchronize the execution of individual processing elements centralized or distributed controllers are used. Controllers are implemented as counters with predefined terminal counter values. In order to create a pipeline, registers are added to the output of processing elements with shorter latency than relevant processing elements in the same DGF node. Here it should be noted that this design approach uses fixed data paths and controllers and therefore lacks configurability and programmability.

A data path is generated for multiple applications, merged and optimized [41]. Data path synthesis is done based on application DFGs. If a new application needs new data path elements, they will be added. This suggests that these new applications would need to be from the same application domain (DSP filters for example). The addition of new PEs is done via FPGA reconfiguration memories (bit map update). A lattice like interconnect is recommended since it has a good chance of successfully taking new PEs.

Requirements for successful implementation of a frequently used DSP reduction operation over large data sets are given by Lin et al. [21]. The considered reduction operation is made from a number of simpler and separate operations. In order to speed up the processing time it is required to implement operations concurrently. These concurrent operations may have different processing latencies thereby requiring synchronization. Concurrent operations are synchronised using buffers. This paper does not go over FPGA reconfiguration and programmability, however it provides a very interesting analysis of DSP reduction operation and buffer based synchronization of concurrent operations.

2.4 FPGA-Based Coprocessor Elements

In this section we briefly describe the main elements of the coprocessor architecture developed in this thesis.

2.4.1 Interconnect Architectures

In this chapter we look into two major interconnect architectures: a crossbar and a 2-D mesh [83].

There are a number of crossbar switch architectures [84]. Most of them belong to one or a combination of the flowing: Input Queued (IQ), Output Queued (OQ), Combined Input and Output Queued (CIOQ) or buffered crossbars. Furthermore message passing from source to destination ports is controlled by an algorithm. Algorithms are executed by distributed hardware modules, often located in source and destination ports. These separate hardware modules communicate to each other by request/grant protocols. Algorithm examples are Input-Output Round Robin (RR-RR), Longest Queue First (LQF) and Oldest Cell First (OCF) [85].

One of the challenges crossbar designers are faced with is maximizing the number of input to output connections. Crossbar scheduling is done in steps. In the first step all input port schedulers select one of the output ports they have requests for. Similarly the output schedulers select one of the inputs from the first step and acknowledge their selection back to the input schedulers. In the final step input schedulers select one of the acknowledged ports from the step two. Chances that output schedulers will select the same input are realistic. Therefore the crossbar throughput for non-uniform traffic can never be 100%. Making each output scheduler aware of other output schedulers theoretically would help but in practice is hard to accomplish. iSLIP is a well-known industry used scheduling algorithm.

It should be noted that most SoC messages sent through interconnects (Network-on-Chips or NoC) (requests, responses or packets) require correct ordering at the destinations. NoCs, most of the time, cannot guarantee message ordering from different sources to the same

destination. To do so they would become very inefficient due to frequent Head of Line (HOL) blocking.

Message ordering at the destinations is established by the destinations themselves. These destinations are often complex coprocessors with large amount of memory available for reordering.

In this thesis the message ordering is done by a centralized interconnect controller. Having a centralized controller impacts a selection of crossbar architectures. We prefer a centralized controlled crossbar versus a distributed algorithmic control due to the processing scheduling simplifications.

The following sections present a few crossbar architectures going from simplest to more complex.

2.4.1.1 Crossbar

2.4.1.1.1 Basic Crossbar

The basic crossbar just simply connects every source (S) to all destinations (D). As soon as Source 0 (S0) is done processing it forwards results to the relevant destination (one of destinations D0 to Dm).

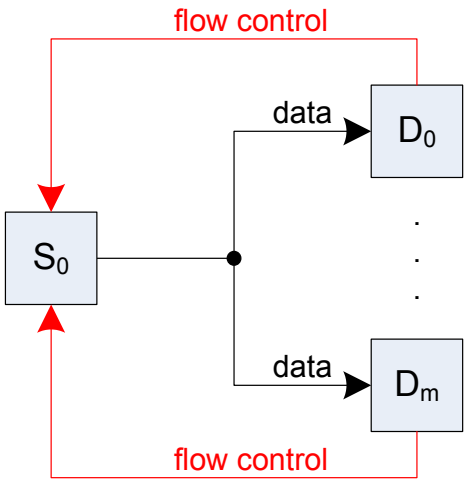

Figure 4 – Basic Crossbar

The main disadvantage with this crossbar is that any Destination may flow control the whole system. This head of line (HOL) blocking case occurs if for example D0 asserts flow control to S0 and if S0 has data destined to D0 at its head. The destination Dm is blocked as well.

2.4.1.1.2 Crossbar with Virtual Output Queuing

N x M crossbar is a structure interconnecting N inputs to M outputs. Source nodes support Virtual Output Queues (VOQs). They can have messages for more than one output M. There is one VOQ per destination. VOQs prevent HOL blocking condition.

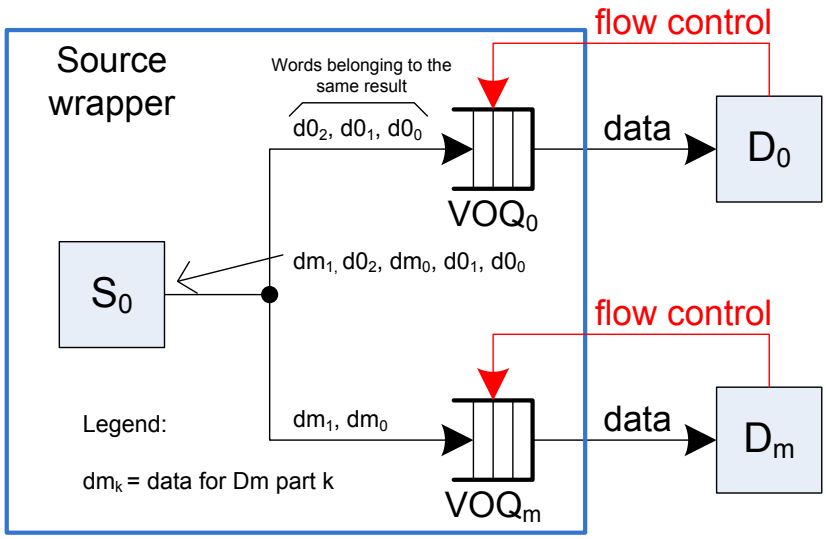

Figure 5 - VOQs

From Figure 5, the Source Processing Element S0 provides interleaved back-to-back data words to any of destinations D0 to Dm. If any of the destinations periodically flow controls S0, they would only flow control its own VOQ – other destinations are unaffected.

Source processing elements could process a number of operands and send complete single-cycle or partial multiple-cycle messages (results) to required destinations connected to the crossbar output ports. The multiple-cycle messages for a destination could be interleaved with messages to different destinations. Assume the source supports multi-threading. Source ports do not need to wait till a complete result is evaluated and head of line blocking other processing activities.

This source multi-threading sounds promising but comes with the following problems:

- Operation dependencies are impossible to preserve. From the above figure Do and Dm may complete their processing at arbitrary moments.
- The crossbar latency due to the multi-step scheduling is neither predictable nor consistent. As an example, consider that D0 and Dm provide results to another processing element Dy. Dy scheduling by a centralized controller is not trivial.

A gross effect of the above two issues is that the VOQ processing time and latency may actually be worst versus a more plausible solution using a simpler crossbar.

2.4.1.1.3 Non-Buffered Crossbar (Processing Element Crossbar)

In this section we present crossbar architecture acceptable for centralized control via the Electron and having Processing Elements as sources and destinations. This crossbar does not use VOQs for the reasons described above. Instead each destination can capture one word per source. Therefore if a destination needs two operands, it could capture a low latency operand and wait until the second operand becomes available. It offers simple scheduling, good performance and resource utilization.

The following figure shows a 4x4 crossbar [83]. Outputs from P0 to P3 (source ports) can be connected back to inputs P0 to P3 (destination ports). As an example, consider the case where inputs P0 to P3 output data for P3. Therefore P3 can select operands from any input – including its own output. Note that cross-points, shown as circuits are not of physical nature (routers for example), they just represent busses connecting source and destination ports.

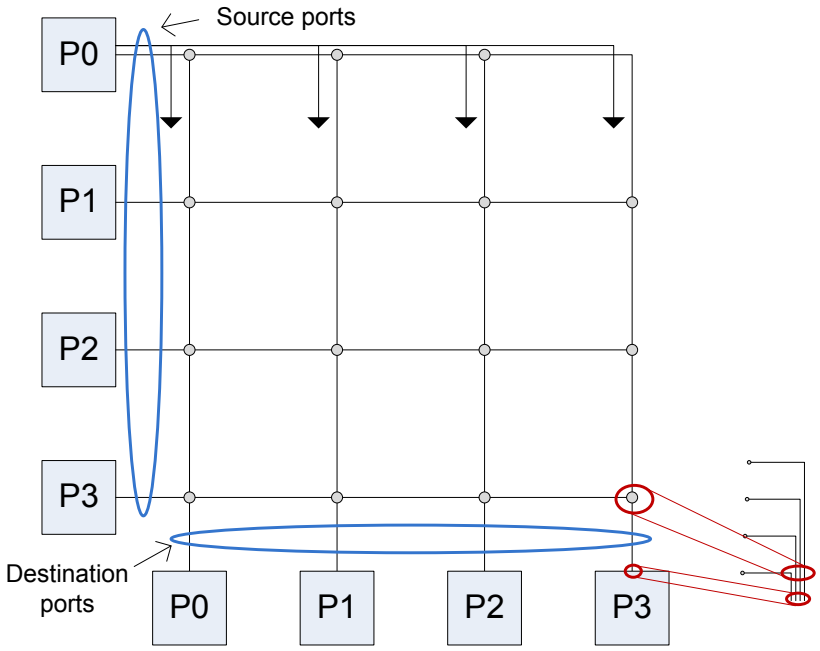

Figure 6 - Crossbar

In order to reduce a number of crossbar wires, a sparse crossbar may be an acceptable solution (Figure 7). In this crossbar, output of processing elements cannot be connected to themselves. Rather a local connection takes care of looping back a PE output to the input. The sparse crossbar greatly simplifies the crossbar. The 4x4 sparse crossbar has 12 rather than 16 cross points versus the non-sparse crossbar.

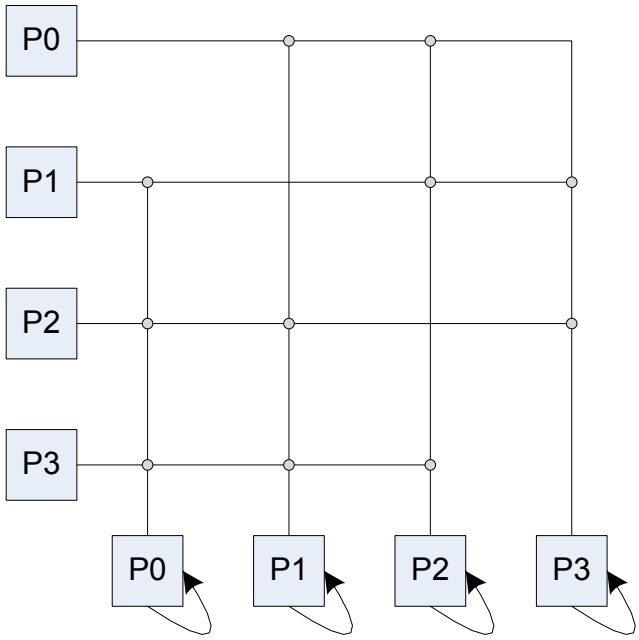

Figure 7 – Sparse Crossbar

Due to the operation dependencies the scheduling needs to be done from one centralized place.

These two crossbars could work with multi-threaded PEs but the PE multi-threading is inefficient.

2.4.1.1.4 Buffered Crossbar

Next we look into high performance buffered crossbars [86][87][88][89]. The buffered crossbar has FIFOs in all crossbar cross-points. The buffered crossbar scheduling is done in two independent steps making the port assignment scheduling much simpler than the crossbar with VOQs. In the first step each source port selects a non-full cross-point FIFO and writes to it. In the second step each destination port selects a non-empty cross-point FIFO and reads from it. Therefore all input and all output schedulers work independently and in parallel to each other. All schedulers are Round Robin based. Figure 8 is a full and Figure 9 a sparse buffered crossbars. Note that resource saving offered by the sparse buffered crossbar could be significant. Each removed Cross Point Connection is a FIFO.

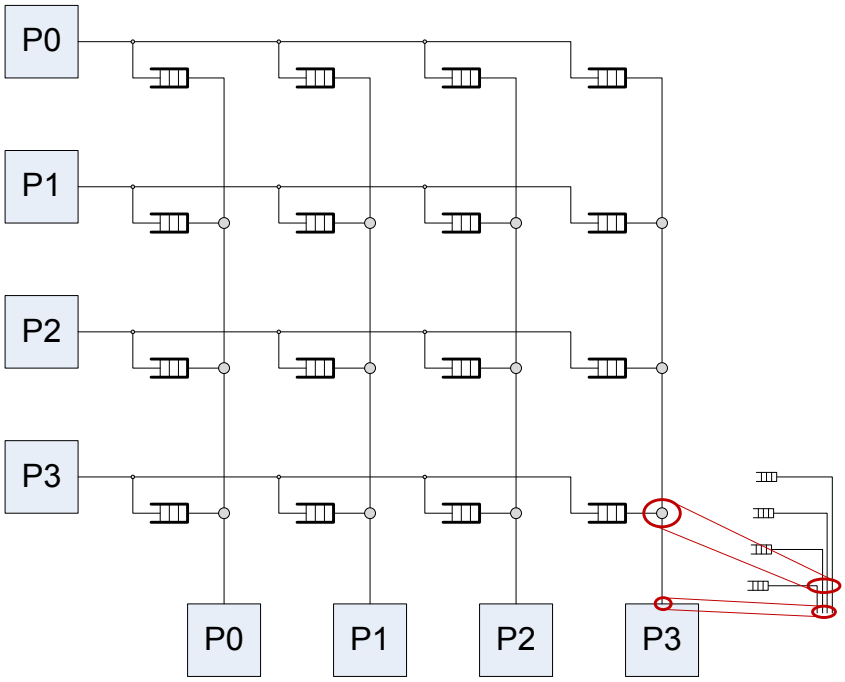

Figure 8 - Buffered Crossbar

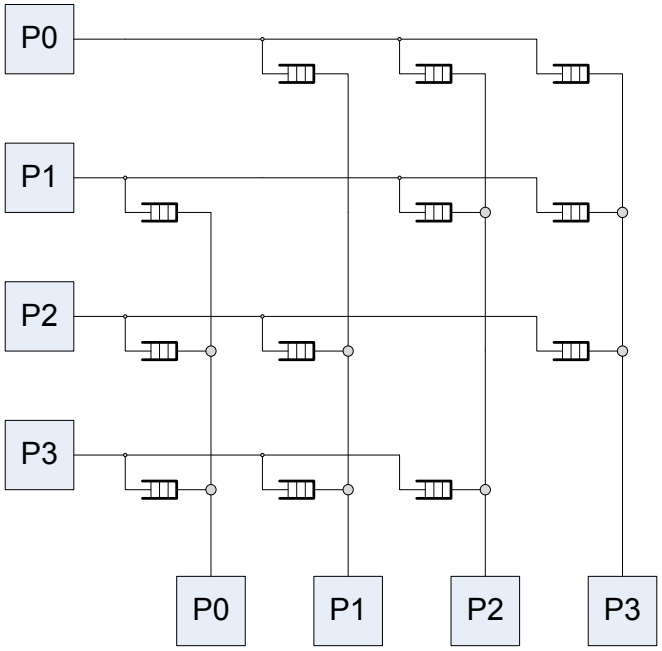

Figure 9 – Sparse Buffered Crossbar

The buffered crossbar would work well with multi-threaded PEs. However input and output schedulers can no longer be Round Robin. Due to the operation dependencies the scheduling would need to be done from one centralized place.

2.4.1.1.5 Crossbar Summary

In the previous text non-buffered and buffered crossbars are introduced, including their sparse crossbar versions. The sparse crossbar optionally could use local loopback within PEs.

Which crossbar type to select depends on the PE latencies. If all Processing Elements have the same or similar latencies the non-buffered crossbar will perform well providing the similar performance as the buffered one – but with lower implementation costs.

A destination Processing Element may require two inputs to perform a calculation task (an addition, multiplication). If two source Processing Elements have different latencies at the destination Processing Element, the buffered crossbar may improve the system throughput. Low latency source Processing Elements could perform a number of calculations and push results to cross point FIFOs belonging to a number of destination PEs. In the non-buffered crossbar, the low latency source PE would need to hold the current output so it could be latency matched with a high latency source Processing Element at the destination PE.

In the practical part of this thesis we opted for an output buffered sparse crossbar implementation due to its scheduling simplicity and low logic resource requirements. The throughput was not minimized. The long latency PEs, if operation dependencies would allow it, are scheduled first. In our crossbar implementation all PEs have one word deep FIFO per input port. Low latency input can be captured and held until the second operand is available.

2.4.1.2 2D-mesh

Crossbar based interconnects may present routing issues for a larger number of input and output nodes. An alternative comes in a form of 2D mesh structures [83][90][91][92]. 2D mesh offers more pipelining therefore easier backend timing closure.

A typical 2D 4x4 mesh NoC (Network on Chip) is shown in Figure 10. The 16 NoC processing elements (shown as square blocks) are interconnected with routers. Each router (shown in the circle on the right side), except the edge routers, has 4 input and 4

output ports allowing any router output to be connected to any input. NoC processing elements can be connected to any other NoC element via a number of paths, each path having different delays.

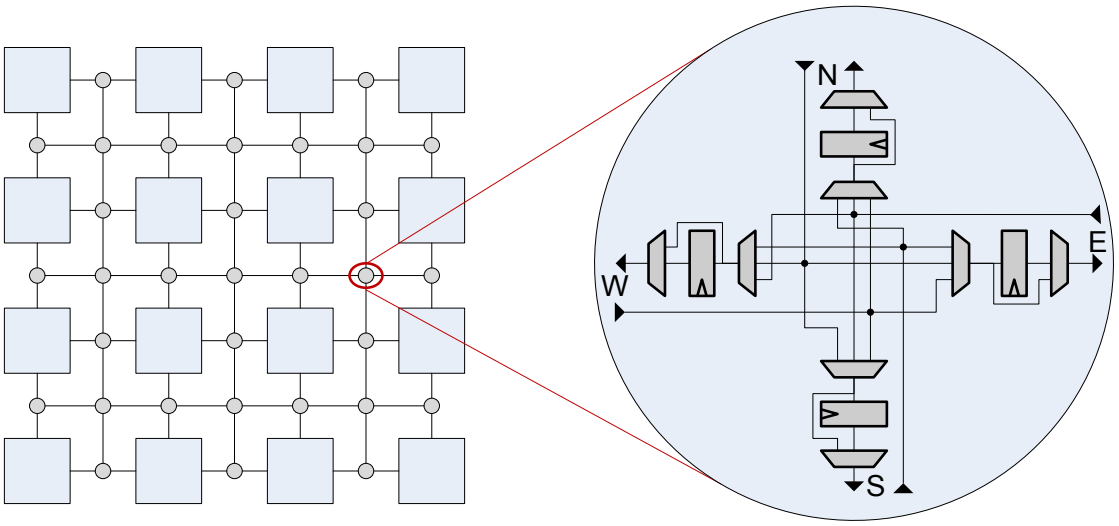

Figure 10 - 2D-mesh

Figure 11 depicts a variety of connections available through the NoC. Operands OP10 and OP11 are brought to the processing element M2. Operands OP00 and OP01 are routed to M1 in a similar way. M1output is routed to A1. Output from A1 and M2 are connected to A2.

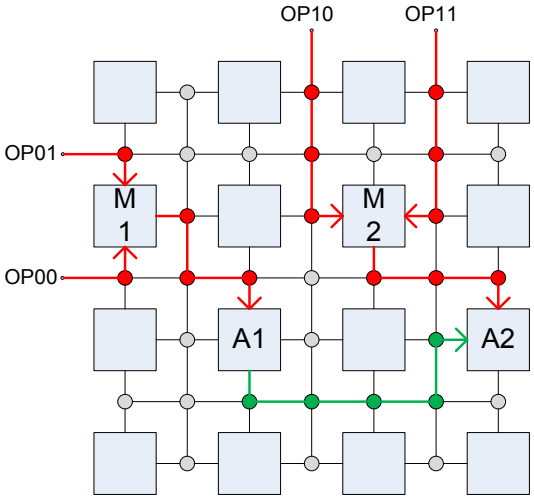

Figure 11 - 2D-mesh, Information Flow

Figure 12 depicts a 2D torus mesh. In this NoC architecture speed of boundary connections is improved by providing extra sets of wires.

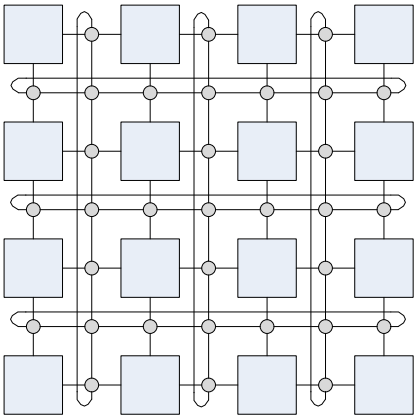

Figure 12 - Torus 2D-mesh

2.4.1.3 Crossbar versus 2D-Mesh

2D mesh in its nature is a pipelined (a regular) structure therefore presents less chip backend routing and timing issues than the crossbar. However 2D mesh introduces extra clock cycle latencies between source and destination ports versus a crossbar.

In this project a 2x2 2D mesh is used to interconnect coarse grained processing Kernels while the crossbar is used within Kernels (Electron instantiation) to interconnect Processing Elements. Crossbars required by Electron are 5 x 5 or under therefore the backend routing and timing closure are not an issue.

2.4.2 Controller

The controller needs to provide all control signals required by the interconnect structure, Processing Elements, register file and other system components. The number of control signals required depends on the interconnect size and type and could be in a few dozens. The Very Long Control Word (VLCW) architecture is capable of having this many control signals. Main VLCW features [93] are:

- The system is controlled by one control unit. This control unit issues one VLCW instruction per clock cycle.

- Every VLCW instruction is composed of a number of independent yet coupled operations.
- Operations within VLCW instruction execute in a small and predictable number of clock cycles.
- For the type of VLCW architecture being considered here we allow the compiler to have full access to all hardware components – therefore making hardware less complex. Simpler hardware allows for more controller instantiations as well as lower power consumption.

Sections 4.4 and 4.5 describe in details the controller named Electron developed for this thesis.

2.4.3 Compiler

The compiler is a software tool capable of:

- Parsing the application program looking for opportunities to offload the program execution. This parsing engine could rely on the compiler "hints" embedded in application source code (pragmas) by the programmer and/or look for the opportunities on its own.
- Creating a Data Flow Graph (DFG) or a pseudo-code of the application code that requires acceleration.
- Translating the above extracted information into a "cost function". This cost function is used to determine if the application code offload would be beneficial. The compiler needs to compare execution time on the CPU and FPGA. Execution estimations are not straight forward. For a server, this time is influenced by the CPU type as well as the clock rate, and the amount of available data/instruction cache. The size of the CPU's cache is a very important parameter. Cache size versus size of the data that require processing will determine data cache trash rate.
- Considering how often the calculation is performed. If many application program runs are required, costs of partially reconfiguring the FPGA by new component from Virtual Hardware library perhaps are acceptable.
- Creating the code required by the FPGA and downloading the code to the FPGA.

- Creating functional calls for the application program and interfacing those with the FPGA driver.

2.4.4 Memory Subsystem

Memory Subsystem throughput is defined by the memory hierarchy. This hierarchy is made from:

- Host memory (PC system memory)
- Host to FPGA interconnect (PCIe, Ethernet)
- FPGA board memory (DDR/2/3)
- FPGA device memory slices mapped to FPGA BRAMs

This memory subsystem needs to be designed so it does not limit the FPGA processing speed. How much memory bandwidth is required depends on the FPGA processing capability – determined by the clock rate, FPGA degree of concurrency (number of parallel processing engines - kernels) and the rate of the memory references. If processing kernels take multiple clocks to complete their workload with fewer memory references the input data bandwidth requirements are lower.

When possible, repeated memory references should be done to a faster memory system sub-component (BRAM versus DDR).

In this project we use Coarse and Fine grained Kernels (see Section 4.7) where data is moved from centralized FPGA memory to processing Kernel's General Purpose Registers (GPRs – flip-flop based memory structure) for lower latency access. The centralised FPGA data memory is populated via the PCIe link. A similar model is used in streaming processors [94].

2.5 Server Operating System Jitter

Operating System (OS) jitter is variability in an application processing run time [95]. For the same application workload, for example a series of calculations, the application run time would complete in variable and unpredictable time. Reasons for this behaviour are

many, for example background daemons waking up, other application events or hardware interrupts.

The problem of the unpredictable run time comes into play if an application is running over 100s or 1000s or processor cores. The slowest CPU core determines the speed of the whole system – literally the slowest core determines the system processing speed.

Therefore minimizing the Operating System caused application run time variability is of a high importance.

2.6 Power Consumption in Integrated Circuits

Integrated circuit power consumption comes from three components [96]: static, dynamic and IO (Input Output) power.

Static power is the power being consumed when no signals are toggling. This power component is caused by the leakage current and in modern chips it is a major power consumption source. Static power is directly proportional to the circuit voltage supply, transistor voltage threshold and the transistor technology.

Dynamic power is the power consumed due the signal toggling. As a consequence logic gates are charging and discharging capacitive loads. At the same time logic gate transistors are creating short circuit currents. Dynamic power can be described with the following equation:

$$Pdynamic = \left(\frac{1}{2}\mathrm{C}\,V^2 + \mathrm{Qsc}\,\mathrm{V}\right) \times \mathrm{F} \times \mathrm{A}$$

Where C - capacitive load, V – circuit voltage supply, Qsc – transistor short circuit current, F – operating frequency and A – signal activity rate.

The IO power is the power consumed by off chip drivers, high speed transceivers and other general purpose IOs.

2.7 Amdahl's Law

Amdahl's law defines how a system can be improved if only a part of this system is improved [97][98]. In parallel computing Amdahl's law is used to compute the theoretical system speedup for a given application and a number of processing cores. The law states the system speed cannot be improved beyond the speed of the sequential part of the application.

Over the time since it was published Amdahl's Law sparked many debates related to system performance obtained by multiprocessors built with a few powerful cores or by many small cores. This debate is still going on.

2.8 Implemented Application

2.8.1 Goals

The project goal is to provide enough evidence that the chosen architecture based on Electron processor and a crossbar can provide flexibility and speed required to compete with server CPU based implementation.

The problem at hand is implementation of Black-Scholes equation used in mathematical modeling of financial markets.

2.8.2 Options Trading

Options are contracts giving rights, but no obligations, to one party to perform a transaction with another party following specified terms. The Call option provides the option holder the right to purchase an underlier at a specified price. European option can be exercised only on the option expiration day.

2.8.3 Black-Scholes

Equations used in this project are based on work done by Fischer Black and Myron Scholes followed by work done by Robert Merton [99]. For this work they were awarded the Nobel price. Formulas developed are used to determine the theoretical estimates of Put and Call price of European options over time. High-Frequency trading, trading executed by

computers collocated by trading houses rely on financial models based on Black-Scholes formulas. In this project we implemented the Call price model.

Variables used in these formulas are:

s = the price of the underlying stock

k = the strike price

r = the continuously compounded risk free interest rate

q = the continuously compounded annual dividend yield

t = the time in years until the expiration of the option

σ = the implied volatility for the underlying stock

Φ = the standard normal cumulative distribution function

Call price:

$$C = \mathrm{s} \times e^{-qt} \times \Phi(\mathrm{d1}) - \mathrm{k} \times e^{-rt} \times \Phi(\mathrm{d2})$$

Where:

$$d1 = \frac{\ln\left(\frac{s}{k}\right) + (r - q + \left(\frac{\sigma^2}{2}\right)) \times t}{\sigma \times \sqrt{t}}$$

$$d2 = d1 - \sigma \times \sqrt{t}$$

d1 = number of standard deviations from the mean of the distribution of stock prices. It is a measure of the spot price relative to the mean stock price.

d2 = the strike price relative to the mean stock price

Formula for the standard normal cumulative distribution function [100] (note that it is sequential in nature, see Section 5.2.2):

$$\Phi(d) = 1 - \phi(d) \times (b1 \times t + b2 \times t^2 + b3 \times t^3 + b4 \times t^4 + b5 \times t^5)$$

Where:

$$t = \frac{1}{1 + (b0 \times d)}$$

ϕ is the normal standard distribution is defined as:

$$\phi(d) = \frac{1}{\sqrt{2\pi}} \times e^{-\left(\frac{d^2}{2}\right)}$$

or:

$$\phi(d) = rsqrt2pi \times e^{-\left(\frac{d^2}{2}\right)}$$

Constants:

b0 = 0.2316419,

b1 = 0.319381530,

b2 = −0.356563782,

b3 = 1.781477937,

b4 = −1.821255978,

b5 = 1.330274429

rsqrt2pi = 3.98942291736602783203125E-1

2.9 i5-3570 CPU

In this thesis the FPGA performance is compared with Intel i5-3570 CPU. This CPU is a superscalar processor. It has 4 cores with 2 threads each. It is an Ivy Bridge running at 3.4GHz (3.8GHz Turbo Boost), designed in 22nm process node. The CPU has 6MB L3 cache, 4 x 256KB L2 cache (256KB per core), 4 x 32KB of data and 4 x 32KB of instruction cache

(32KB per core). There is a SIMD unit therefore i5-3570 supports streaming extensions. In addition it supports encryption and vector extensions.

Being a superscalar i5-3570 can execute more than one instruction at the same clock cycle. Instruction scheduling is dynamic – the CPU hardware schedules instructions. Executing multiple instructions require the CPU to have a number of execution units, such as Arithmetic Logic Unit or a Floating Point Unit.

Due to the dynamic scheduling, i5-3570 consumes more power than simpler statically scheduled processors, such as VLIW processor.

2.10 Development Board

ML605 is a Xilinx Virtex-6 based FPGA (XC6VLX240T-1FFG1156) Evaluation Board. The board can be used as a standalone or as a PCIe slot plug in card in PCI Express® Gen1 8-lane (x8) and Gen2 4-lane (x4) modes. In this project the board was used as Gen2 4-lane. In this configuration the PCIe interconnect and Xilinx FPGA PCI core can deliver effective 16Gbps. This effective throughput number needs to be reduced by the transaction layer protocols.

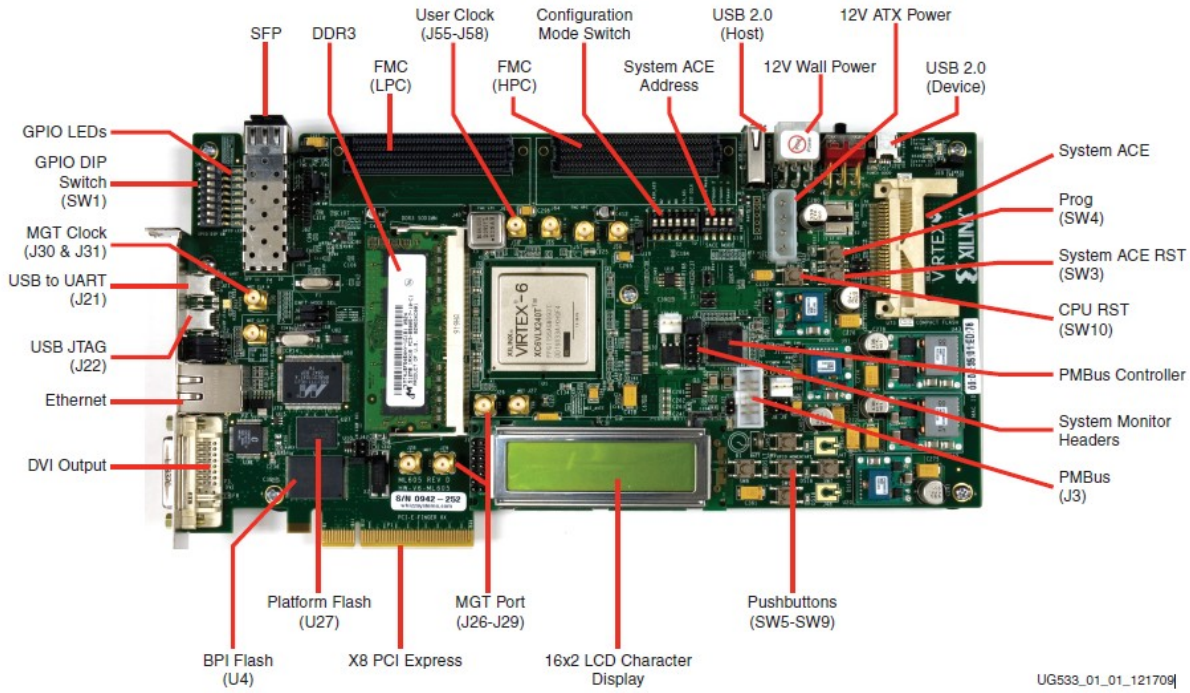

Figure 13 – Xilinx ML-605 Development Board

During the hardware and test software development a debug software tool called PCITree[16] was used. Worth noticing, the PCITree works only on Windows 32-bit.

2.11 PCIe core and Software Driver

In the practical part of this thesis we use Speedy PCIe core and a Windows software driver. Both IPs are developed by Ray Bittner from Microsoft Research [101][102] and are freely distributed for research purposes. The PCIe core is designed for Xilinx ML605 development board.

2.11.1 Speedy PCIe Core

Speedy PCIe FPGA core (or "the core") interfaces with Xilinx PCI core on one side and User Logic on the other side. The core provides all functionality required to handle PCIe protocols. It defines an interface to the user logic. From Figure 14 note the 3 main modules:

1. Xilinx PCIe Core
2. Speedy PCIe Core and
3. The "Transaction Controller and Processing Complex". This is a custom design developed for this project.

[16] Available from http://www.pcitree.de/download.html

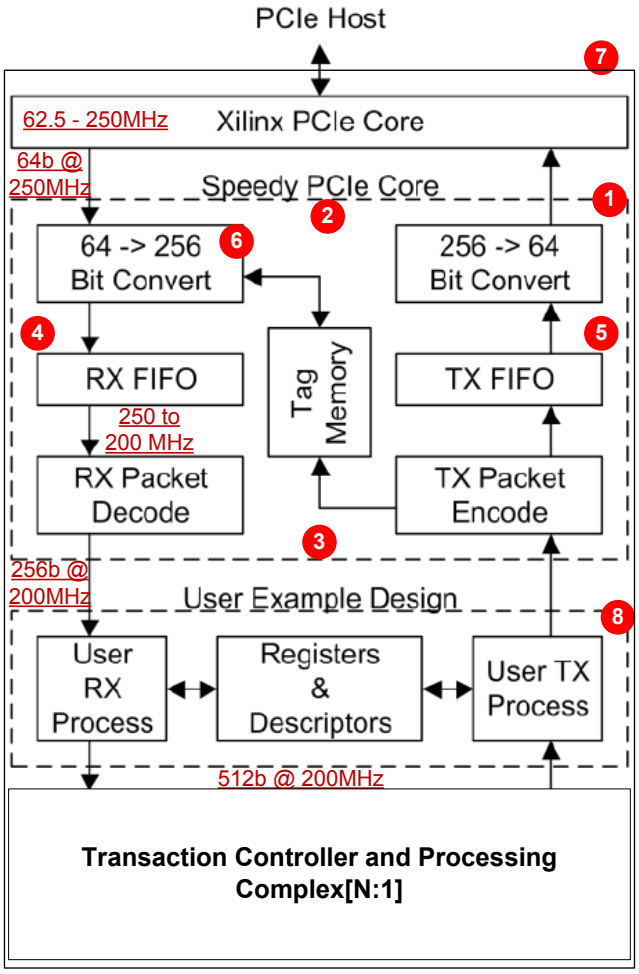

Figure 14 – Speedy PCIe topology

The "Transaction Controller and Processing Complex" replaced Speedy PCIe provided example design that was interfacing with the external DDR3 ML605 based memory.

Besides providing the PCIe related features, the Speedy PCIe Core was designed:

- To be flexible so varieties of host system offload research could be done,
- For easy backend (synthesis, meeting timing, place and route) and
- Speed compatibility with future PCIe standards.

The above requirements resulted in Speedy PCIe architecture with a very wide datapath of 512b and low clock rate of only 200MHz. For information on how this low clock rate affects throughput numbers obtained in this thesis see sections 5.2.6.1 and 6.4.

2.11.2 Speedy PCIe Software Driver

Speedy PCIe comes with a Windows software driver in a form of the source code and precompiled binaries. In this project only 32-bit drivers were used.

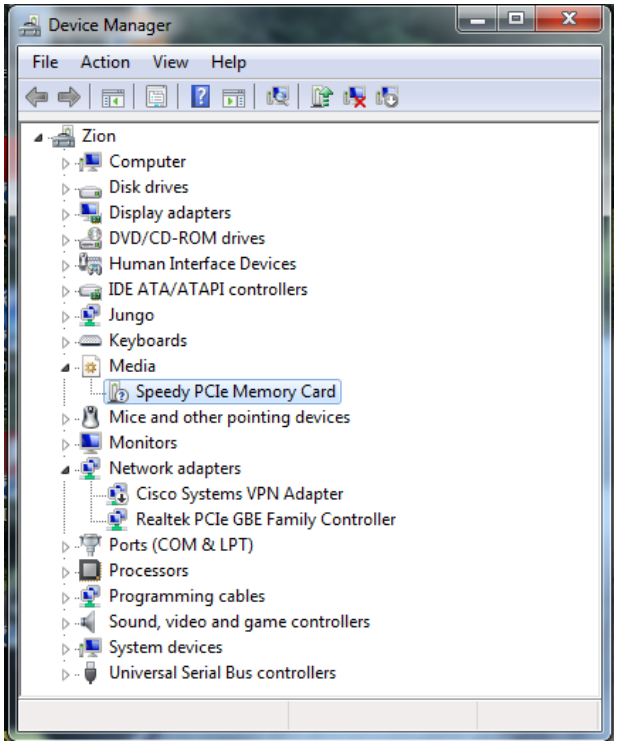

Figure 15 – Device Manager

C++ test programs are provided by Speedy PCIe suit – a useful start point for custom test program development. The test programs write and read to the external DDR memory using the following memory map:

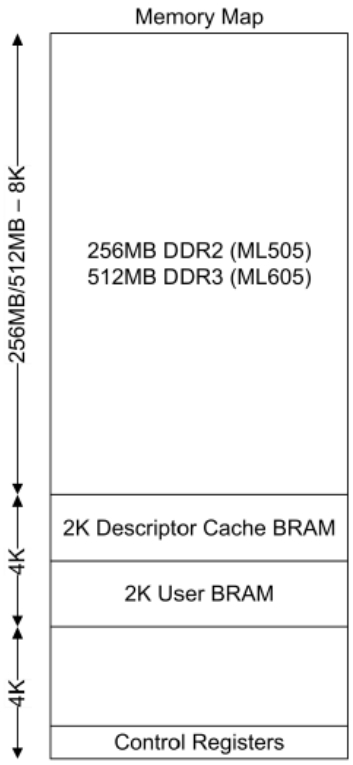

Figure 16 – Speedy PCIe Example Design Memory Map

The Transaction Controller FPGA module is mapped to DDR3 memory space; while it replaced DDR3 related FPGA modules. The provided test programs are substantially modified to fit this application.

In Section 6.1 we describe some of the driver's limitations such as the "read" blocking.

Chapter 3 - Requirements, Thesis Scope and Contributions

Clouds are major power consumers. We would like to determine if heterogeneous computing can bring adequate alternative and lower the energy consumption than servers as well as what applicable technologies, hardware architectures and methodologies would enable heterogeneous computing in Clouds.

The energy equation:

$$\text{Energy} = \text{Power} \times \text{Time}$$

Therefore in order to reduce energy consumption this new technology need to use less Power and/or/optionally run faster than server CPUs. However the effect of Time is twofold, the first one is energy savings and the second one is the Cloud throughput. The throughput directly affects Cloud's ability to service more consumers – and this translates to the Cloud bottom line.

In this thesis it was identified that at least 20% of applications running in Clouds could be offloaded from CPUs and potentially accelerated by FPGA technology. These applications are parallel in nature and have high computation-to-communication ratio. The computation-to-communication ratio determines what applications can take advantage of concurrent processing done on FPGAs [6]. FPGA coprocessors need processing data downloaded from the host system to the FPGA. This time represents the communication part of the computation-to-communication. Therefore we need to increase the computation part using coarse-grained architectures.

FPGAs consume less power than server CPUs or GPUs technology. For this thesis the FPGA technology is selected due to its lower power consumption, speedup offered (parallel processing) and flexibility (full and partial reconfiguration). To utilize these FPGA features we still need applicable hardware architectures.

In this section the following topics are included:

Analysis of the literature and requirement definition – This section is based on the Background section. Benefits and disadvantages of a number of FPGA design methodologies are provided. Methodologies are compared. Next we define a set of requirements for successful deployment of FPGAs in Clouds. What follows is analysis of FPGA coprocessor architecture that overcomes all deficiencies of the current FPGA design methodologies.

Thesis Scope – In this section we list all steps required to enable use of FPGAs in Clouds and what steps are covered by the thesis.

Contributions – Here we list thesis contributions to the current state-of-the-art.

3.1 Analysis of the literature and requirement definition

3.1.1 Methodology Comparison

In Section 2.3 we analysed a number of FPGA design methodologies. The analysis considers Clouds as the computing environment. Cloud computing could be greatly enhanced by reconfigurable computing through improved performance and lower power consumption. FPGAs offer both features.

To improve the application programmer's experience the goal is to have a single programming environment where CPU and acceleration hardware would be programmed at the same time from high level languages such as C. Although it is hard to argue with the single programming environment goal, the technology required to get there is not immediately clear.

Custom designs offer the best performance and best power utilization, but with the expense of poor flexibility and long development time.

Embedded RISC FPGA cores offer unparalleled flexibility but with the expense of power consumption and lower performance due to their fetch-decode-execute architecture.

The concept of Virtual Hardware offered with pre-synthesised hardware templates opens new possibilities for a faster device "update" via a partial reconfiguration (Section 2.3.3).

High Level Synthesis appears to be great in what it is invented to do – create highly optimised, sometimes larger than expected, fixed function circuits. If they are to be used in the Cloud environment, the FPGA reconfiguration is mandatory. Every new acceleration task would require running these tools, followed by synthesis and floor planning the new circuit and finally device reconfiguration.

The following table gives methodology comparison. Terminology used:

- Electron is architecture developed in this thesis
- RC – Reconfiguration
- FPGA RC- Complete device reconfiguration
- PRC – Partial reconfiguration. Depending on what is being reconfigured, FPGA sections are not operational.
- Synthesis, P&R – FPGA backend steps, time consuming. A new FPGA load is being created. This step is not suitable for Cloud application.
- Virtual HW – Does the methodology offers use of per-synthesised HW templates. Switching in a new template requires partial reconfiguration only, but not Synthesis and P&R steps.
- Multi-core – is the methodology feasible for multi-core processing support
- Operational during RC – If device is being reconfigured is it out of service
- Flexibility – how versatile is a methodology
- (V)ery (P)oor, (P)oor, (G)ood, (V)ery (G)ood, (M)aybe, (O)ptional, (Y)es, (N)o

Table 1 - FPGA Design Methodology Comparison

Design Methodology	FPGA RC	PRC	Synthesis, P&R	Virtual HW	Multi-core	Operational during RC	Flexibility
Custom	Y	N	Y	N	N	Y	VP
HLS	Y	Y	Y	Y	N	M	P
HOS	M	Y	M	Y	N	M	G
NISC	Y	N	Y	N	N	O	G
Electron	N	O	N	Y	Y	O	VG

Electron architecture will be described later. However, from the table we see that Electron is the best architectural choice for Clouds. It offers flexibility, it can create multi-core processing, gives opportunity to optionally add a new Virtual Hardware component and partially reconfigure FPGA. At the same time it does not need FPGA (device level) reconfiguration.

3.1.2 Cloud Coprocessor Architecture Requirements

The solution that offers expected flexibility and speed for the set of Cloud applications listed in Section 1.4 seems to be the use of light processing cores with features similar to those presented in Section 2.3.4.1. These processors would be accompanied by a controllable interconnect and as an option partially reconfigurable data path. They offer coarse-grained data parallelism with features that improve on fully pipelined architectures (see discussion related to Stojilovic et al., [41], Section 2.3.1).

What follows is a distilled list of features that acceleration architecture would need to support for the Cloud computing environment.

System level:

- Do not require FPGA reconfiguration in order to provide processing features when switching in new applications. Rather a fast update of the “processing functionality” such as a program code update is sufficient.

- Allow for optional partial reconfiguration of parts of the FPGA for increased performance.
- Minimize processing latency by activating processing kernels as soon as sufficient data is available on the FPGA (latency hiding). Compare this feature to architectures that first store complete sets of host provided data to a local memory.
- Have the processing time stable and repeatable given the same workload and set of processing requirements (low processing time jitter).

Processing model (see Sections 2.3.1, 2.3.4.1 and 4.8):

- Support coarse-grained data parallelism (computation-to-communication ratio).
- Offer Coarse and Fine grained processing capabilities.
- Support parallel hardware threads (processing time speedup).
- Support locally interconnected processing kernels capable of run-until-completion or sequential processing models.
- Data exchange and kernel scheduling done by a programmable controller.

Hardware architecture level:

- Are capable of controlling parallel data paths.
- VLCW hardware related complexity is passed to the compiler.
- Do not belong to instruction fetch-decode-execute architecture (the RISC family) (*). Rather they belong to “VLIW stream – dataflow control” architecture.
- Do not have a standard RISC instruction set (*). The instruction set requires decoding. This decoded instruction concept cannot well represent numerous possible crossbar states.
- Run a VLCW code containing control words. These control words define the crossbar state (how PEs are interconnected by the crossbar).
- Have operand processing and results forwarding done in the controllable interconnect based data path.
- The data path, made from processing elements and interconnect, allows the data flow processing.

(*) More on disadvantages of RISC processors versus Electron can be found in Section 3.3.1.

3.1.3 Basis of Cloud Coprocessor Architectures

Based on features presented in Section 3.1.2 we define the basis of coprocessor architecture that overcomes all deficiencies from methodologies described in sections 2.3.1, 2.3.2 and 2.3.4.

To the best of our knowledge most FPGA-based coprocessors are created as fixed function designs and are therefore optimized for one application (e.g. a DSP algorithm). Solutions that are flexible are based on RISC processing cores therefore having limited processing throughput. The Cloud-worthy architecture is somewhere between fixed function and RISC architectures with performance similar to fixed function and flexibility of the RISC.

The following is a list of architectures and/or techniques that stand out and whose benefits will be incorporated into the practical part of this project. Rather than giving architectural details, where applicable, links to relevant chapters are provided to avoid text duplication:

The No Instruction Set Computer (NISC) is described in details in Section 2.3.5.1. NISC is used as the basis of the architecture developed in this thesis.

The Streaming Processors are VLIW processor cores. They use a concept of the streaming register file and flexible interconnect within the processing core (ALU) that are of interest.

The variable processing latencies can be addressed by implementing buffers in processing streams. These buffers are used to schedule and clock cycle align streams with different latencies, Section 2.3.5.3.

For the flexible interconnect we consider using a well known and researched Crossbar and 2D Mesh (Section 2.4.1). Both of these interconnect architectures are further optimized for our application. These optimized crossbars minimize GPR port access bandwidth as well as number of crossbar ports.

Hennessy et al. used computation to communication ratio [6] to determine what applications can take advantage of concurrent processing see Section 1.4. They have higher

computation versus communication requirements. Therefore we focus on coarse-grained data parallel architectures.

For the problem at hand we focus on the financial industry [12] [13] [103] [104]. These applications could be offloaded from CPUs and accelerated with FPGAs.

Architectures we are interested in have a set of processing elements connected by a flexible interconnect. This interconnect can be controlled during run-time, therefore we call it a controllable interconnect. The controllable interconnect is made from wires, cross point multiplexers and a cycle accurate controller. The cycle accurate controller runs code that controls this interconnecting structure. The control code is created by a software compiler.

3.2 Thesis Scope

In this thesis we propose a complete framework that would enable use of FPGAs in Clouds. This framework consists of:

1. FPGA hardware architectures.
2. Process of mapping a computation problem into a few computational Kernels.
3. A software compiler capable of creating code executable on proposed hardware architectures.
4. A software compiler capable of creating system level code. This code is executed on proposed hardware architecture and controls and schedules a multi-core system.
5. Process of selecting parallel or sequential Kernel processing model in the case of a multi-core system.
6. Pre-synthesised hardware templates, Virtual Hardware and template libraries supported by FPGA partial reconfiguration.
7. Software drivers required to communicate with the FPGA-based coprocessor and to deal with Virtualization tasks.

Work described in 1 is the main topic of this thesis including the thesis practical part.

Work described in 2, 3 and 4 is well covered in thesis but it is performed manually. All required steps are presented in relevant sections and results/outcome and results of the

manual compilation used in the thesis practical part. Creating a software tool required for these steps would be elaborate and serious undertaking. It would allow an in depth research in optimization techniques that would lead into better system performance.

Work described in 5, 6 and 7 is only mentioned as an integral part of a FPGA toolset. Creating a software tool required for these steps would be elaborate and serious undertaking. Especially, the step 6, before creating such a software tool, lots of research is required to determine what templates (processing elements) would reside on FPGA and what as Virtual Hardware. This research would need to take a close look into Cloud applications of interests such as financial and scientific computations.

3.3 Contributions

Section 2.3 presents a detailed review of the existing FPGA industrial and academic design methodologies. These methodologies were analysed from the speedup offered and flexibility point of view. Based on findings it was determined that two academic solutions presented in sections 2.3.5.1 and 2.3.5.3 are a good base for our further work. This work materialised into a new developed processor named Electron.

Electron is hardware architecture with programmable control path and a novelty in a form of controllable and optionally reconfigurable data path. We call this data path DynaPath. The controllable and reconfigurable data path feature is built from a unique use of a crossbar and the notion of Virtual Hardware (pre-synthesised hardware templates).

Let us now list the contributions here and then describe them one by one.

3.3.1 Processing Pipeline

Figure 17 presents a 6 stage RISC (top) and Electron (bottom) processing pipelines. For a basic 5 stage processing pipeline description see Hennessy et al. [6].

From the figure: IF – Instruction Fetch, ID – Instruction Decode, OF – Operand Fetch, EX-Execution Unit, WB – Write Back (there could be more than one stage) and CS – Code Stream.

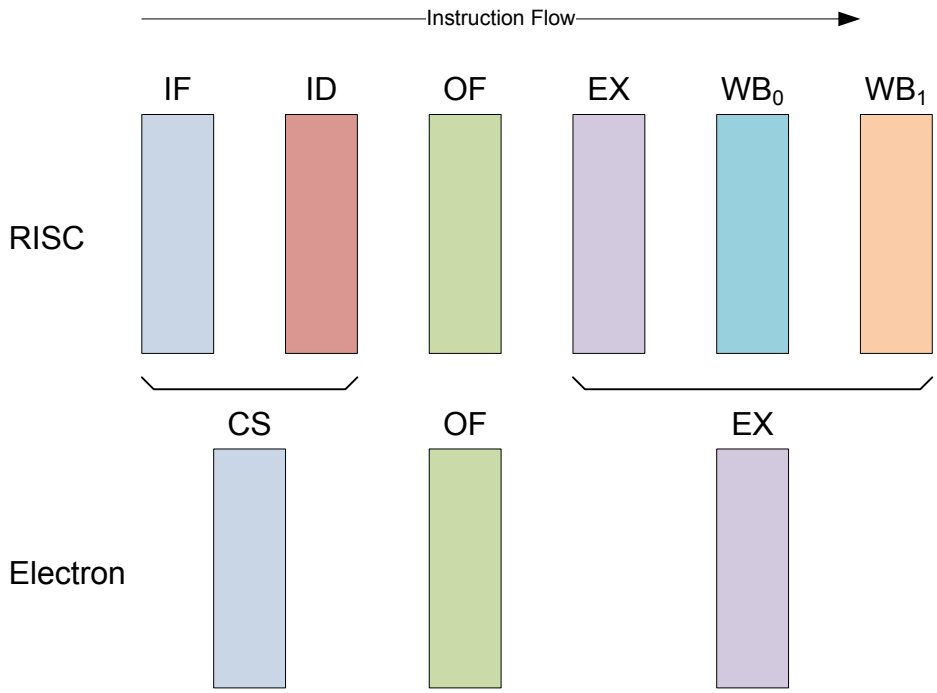

Figure 17 – RISC and Electron Pipeline

Electron has only three stages: Code Stream, Operand Fetch and Execution unit. That is two stages less versus the basic processing pipeline or three/four stages less than advanced/complex RISC pipelines. Fewer stages contribute to the processor simplicity (explained in Section 3.3.2), shorter latency and size of a processor.

Latency

The following example assumes a serialised program section. In this case processing unit needs to wait for a result to be evaluated before the next operation is executed.

RISC pipeline latency is 2 times longer. If an average processing element latency is 8 clocks, Electron/RISC total processing latency made from the processor pipeline and the processing element latency:

$$\frac{\text{Electron}}{\text{RISC}} = \frac{8+3}{8+6} = 78\%$$

Therefore Electron has 22% shorter processing latency. This shorter latency has a cumulative positive effect on the program execution time.

Size

A 6 stage RISC pipeline has 2x more flip-flops than Electron (hundreds of flip-flops) indicating that 2 Electrons could fit in place of one RISC pipeline.

3.3.2 Instruction Stages

Electron pipeline is simpler than a RISC processor's pipeline. These simplifications minimize the Electron processor size due to removed pipeline stages (explained in the previous section) as well as due to complex logical cones associated with these removed hardware resources. The removed logic cones allow Electron to run faster (faster clock rate) than a RISC processor. The following is a list of simplifications:

Electron CS stage in RISC is made from IF and ID stages, Electron and RISC both have OF stage and Electron EX stage in RISC is made from EX and WBs stages.

CS stage streams Electron instructions, they do not require decoding. Electron instruction directly controls processing elements, crossbar, registers and so on. One can see Electron instruction as already being decoded. Decoding an instruction with variable size and location of fields presents a challenge [105]. What is offered with Electron's approach is a processor simplification due to removed ID stage (processor size and achievable clock rate improved) and ability to control multiple hardware modules. It should be noted that some instruction level complexity is left to the compiler.

Electron EX stage is made from a crossbar and all hardware blocks connected to it (DynaPath). The crossbar also provides the write back features. The WB stage is used as a way to pass back data to EX (intermediate results) in addition to write them to GPRs. More WB stages are available more reduced access rate to GPRs since intermediate data can be found in one of WB stages. If GPRs are located in a SRAM, the number of SRAM ports is directly related to the GPR access rate. Number of ports significantly affects the SRAM size. If a processor has more than one data path, this effect is even stronger. If SRAM access rate can be reduced, GPR SRAM can be simpler and smaller. The Electron crossbar has buffers at processing element inputs and outputs therefore GPR access rate is drastically reduced. The intermediate data can be found in the Crossbar – often much longer that RISC WB

stages would. An important fact with RISC using WB stages is requirement to compare operand addresses from OF and WB paths. More WB stages, more complex this logic gets. The pass back feature in RISC is complex, requires logic resources and it is hardware controlled. In Electron this feature is actually under the compiler's control.

3.3.3 Instruction Format

Electron does not use Instruction Set Architecture (ISA); it rather is using Very Long Control Word (VLCW). The VLCW terminology is developed for this thesis. ISA uses one instruction, two operands and one result destination to present only one unit of work per clock cycle (for example – ADD, LOAD). The Electron data path can do a number of units of work per clock cycle. At the same time ISA requires use of an Instruction Decode (ID) unit. ID adds one stage to the processor making pass-back decision more complex.

VLCW directly controls a number of data flows inside the crossbar. Electron VLCW due to the available hardware (Crossbar and Processing Elements) can easily be extended to compound type of instructions, such as to perform a for-loop multiply-add.

3.3.4 Crossbar Architecture

The crossbar adds a classical processor pass-back feature and operates as the Execution unit.

A number of crossbar architectures are analysed looking for an optimum solution good for throughput, logic resources and backend timing. Two types of crossbars are selected and unique features added. There is one for Processing Tiles (2D mesh) and one for the Electron (crossbar). Unique features:

- In the crossbar a local loopback at the destination port was added. This in effect allowed for the crossbar simplifications (a sparse crossbar).
- Dataflow processing is supported.
- Use of hardware templates (Virtual Hardware) via partial reconfiguration

3.3.5 Electron versus NISC

Gajski et al. suggested a No Instruction Set Computer processor NISC [66]. Electron uses the main concepts from this work but adds new ideas to it:

- A crossbar was added as the main processor data path interconnect. The crossbar connects all processor resources. The crossbar replaced shared buses used in NISC. Shared buses presents interconnect scheduling issues. In order to improve this situation NISC uses a local PE interconnect, where 2 side by side PEs are interconnected in one direction. Consequently local interconnects may work for one but not for different applications.
- In NISC architecture the designer customizes the data path to a given application. Contrary to NISC Electron provides controllable data path interconnect giving opportunity to the compiler to use Electron processor efficiently for a given application. Passing processor utilization complexities to the compiler is more RISC style – but Electron is not RISC architecture.
- A multi-port register file (General Purpose Registers – GPR) is replaced by 1R1W register file. For small memories adding a port doubles the size of the memory. This modification to NISC leads into smaller Electron implementation. If more than one PE needs to access the GPR Write port, nano-code can schedule write operations sequentially, while the write data waits for scheduling in the crossbar flops.
- Eventual 1R1W register file port collisions are eliminated by using an innovative data path based on a crossbar. The crossbar itself can store intermediate data and forward it where required. In NICS all intermediate data must go to and from the Register File.
- NISC is a no-instruction architecture. Electron is VLCW effectively adding ability to have an instruction, such as WAIT or SKIP.
- Instead of the instruction fetch unit, Electron is using Code Stream unit. The Code Stream unit allows observability into the current and next instruction(s). This feature allows for better branch handling.
- Introduction of Special Registers (SR). NISC stores constants in the processor control code memory, a 38-bit control word with a 16-bit constant that takes 42% of the control word. It is very unlikely there would be more than a few opportunities to use this field per program execution yet this field is part of every Code Memory (CMEM) control word. SR results in control code memory width reduction of 42%.

Often used constants (e.g. 0x0 or 0xffff_ffff) are stored there. If program requires other constants, application dependant, they could be provided via the data memory and stored in GPRs for the program life time.

- Further code space compression is achieved by adding WAIT instruction.
- NISC does not give a clear path to multi-core configurations. Therefore new developed features are:
    - Electron data path allows for cores to exchange data. A number of Electrons can be integrated in a multi-core structure called Processing Tile.
    - Electron can be used as a kernel processor or a system processor. If it is used as kernel processor it executes the application code, if it is used as a system processor it schedules individual kernels from a multi-core configuration for parallel or sequential processing.
    - A novelty versus NICS is ability to use shared Processing Elements. The system controller schedules use of shared Processing Elements.
    - Further to the multi-core configuration, a strategy how to coarsen kernels is developed. This strategy is based on the previously mentioned Computation to Communication ratio.
    - Multi-core configuration may require memory protection. This is accomplished by the system controller controlling how to forward intermediate data from a kernel to a kernel rather than the kernel running the application code.
    - Processing Tiles can be used in Single Instruction Multiple Data (SIMD) mode
    - Processing Tiles and Kernels support Coarse and Fine grained compute models. In addition, Kernels support Instruction Level Parallelism (ILP).

3.3.6 Electron versus FlexCore

The FlexCore is similar to NISC [76]. However we provide Electron improvements over FlexCore as well.

- FlexCore instruction width is 91-bits. Electron's instruction width is 57-bits (Section 5.2.3) therefore Electron provides 38% smaller instruction memory footprint versus FlexCore:

$$\frac{\text{ElectronControlWordWidth [bit]}}{\text{FlexCoreInstructionWordWidth [bit]}} = \frac{57}{91} = 62\%$$

- FlexCore is using a 32-bit immediate value. Rather Electron has a choice of using immediate values from Special Register storage or overloads an instruction with an immediate value. Often instructions need a “1”, “all ones” or a “0” values to perform certain calculation. On some other instructions if an immediate value is used, there is no need to provide variables such Register File (GPR in Electron architecture) addresses. This provides an opportunity to overload address variables with an immediate value. Lastly as the last resource Electron can insert an instruction that provides only an immediate value. Only in this case a clock cycle gap is created.
- It was noted above that FlexCore code memory requires large footprint. The footprint comes from the instruction width and so called “static code”. The static code is required due to FlexCore, just as Electron, being a clock cycle accurate. If functional units are processing data and results are not available yet, FlexCore needs static instructions. It was already determined that Electron instruction width is 38% smaller. For the static code problem, Electron supports “Wait” instruction. This is an “idle” instruction being executed by a programmable number of loops – therefore it takes only one code memory word.
- It was noted above that FlexCore code memory requires large instruction bandwidth. This requirement comes from the clock cycle accurate nature of FlexCore, just same as Electron. An instruction is needed per clock rate to control state of the crossbar. Electron solves this problem by:
    - registering and latching DynaPath control signals while in “Wait” instructions are executed effectively minimizing instruction bandwidth
    - by utilizing highly efficient Code Stream code memory unit
- FlexCore uses full crossbar. Every functional unit can be connected to any other, including itself, over the crossbar. Electron DynaPath crossbar used is a sparse crossbar. Processing Elements can be return data to themselves via an internal link. This internal link can also be used for reduction type operations such as accumulate.

- FlexCore does not use any storage within the crossbar. Therefore variable processing latency issue need to be solved by the Register File or Load/Store unit. This approach may create port contention issues.
- Multiprocessing architectures are not addressed at all. Electron offers Processing Tile.
- Shared Processing Elements are not used.
- No concept of memory slices.

3.3.7 Electron and Variable Processing Latencies

Hong et al. suggested use of buffers in systems where individual hardware modules have different processing speed and/or latency [77][78][79][80][81]. This work was adapted, modified and added to Electron crossbar:

- The Electron crossbar implements buffers at its input ports. Buffers work as Virtual Output Queues (VOQ). However Electron does not need buffers to synchronise processing elements since Electron is clock cycle accurate, it knows when data will appear at a node. It rather uses shallow buffers to achieve better utilization of processing elements. Faster processing elements could be used a few times while slow ones are still working on previous data. This approach is of a particular significance if a processing elements need to capture and hold to an operand from a fast processing element and wait for the second operand while the slow processing element is still busy.
- Results from all Processing Tiles are synchronized by implemented buffers before using DMA transfer to the Host system via PCIs link.
- Using deep buffers, introducing flow control mechanism (XOFF or credit based) may introduce jitter in processing time. Therefore Electron uses a cycle accurate central control and shallow buffers.

3.3.8 Processing Tile

Concept of Processing Tile and shared Processing elements are described in Section 3.3.5.

3.3.9 Experimental Results

In order to confirm experimentally usefulness of this research the following steps are taken:

- System level and Kernel nano-code was developed for Black Scholes equation commonly used in Financial Industry
- Software implementation of Black Scholes equation running on a PC is developed
- Experiments were carried on and Electron and SW results compared.

FPGA-based coprocessors require raw data to be downloaded and results to be uploaded to the host system. A low latency and low interrupt strategies were developed for this purpose.

A number of software tools is required by the above described technology. Actual tools are not developed. Instead tool know-how ideas are considered, and these ideas are used manually to prove their practicality.

Chapter 4 - FPGA-based Coprocessor Architecture

4.1 System Level Architecture

4.1.1 Hardware

In this section we present a plug-in coprocessor card. This card could be added to an existing server residing in the Cloud or a separate coprocessor chassis. In either way, the card supports PCIe as the main interconnects with the outside world.

The following figure depicts the conceptual architecture of an FPGA-based coprocessor.

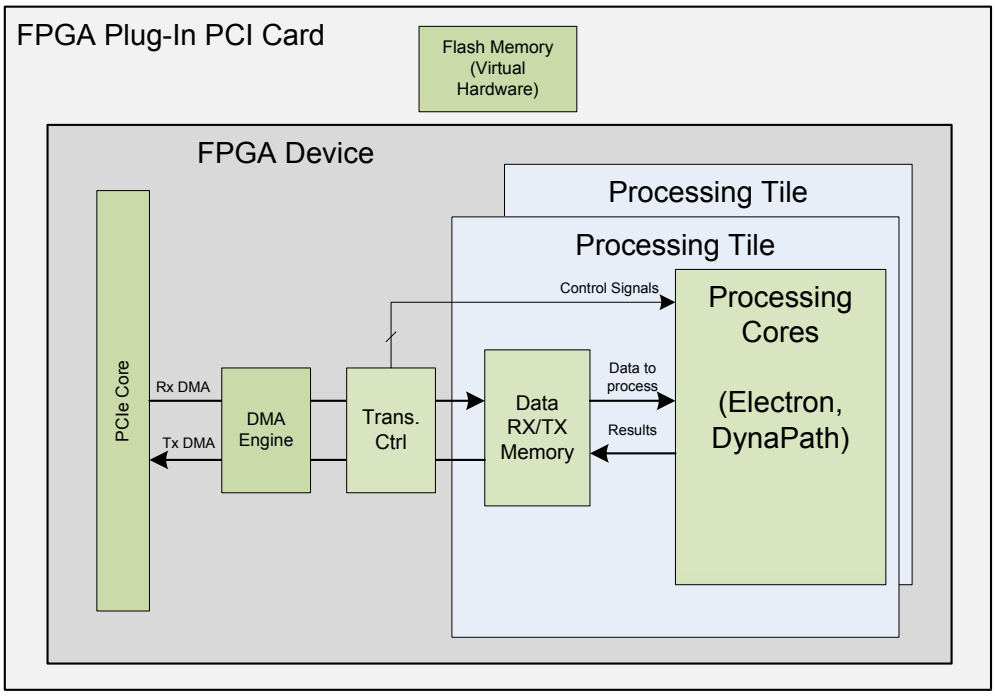

Figure 18 – FPGA Coprocessor

The main FPGA Coprocessor components and their functions are listed below.

- Flash Memory contains Virtual Hardware, a library of pre-synthesised and ready to be used coprocessor hardware components, such as Processing Elements (see list of contributions, Section 3.3.4). Components are used for the partial FPGA reconfiguration.
- PCIe Core and DMA engine offload Processing Tiles from PCIe related transport tasks.
- Receive and Transmit memories are used to store data that needs processing and results obtained by the Processing Tiles.
- The FPGA coprocessor hosts a number of Processing Tiles (PT). Each PT executes processing code concurrently. If all PTs execute the same processing code they provide Single Instruction Multiple Data (SIMD) processing model. PTs cannot exchange data among themselves (PT is listed in contributions, Section 3.3.8).
- Kernels, Electron instantiations, belonging to a PT can operate in parallel or serially. If connected serially they create a processing pipeline, where each Kernel does a part of the processing task. Kernels can exchange intermediate data. If Kernels operate in parallel to each other they provide Single Instruction Multiple Data (SIMD) processing model. Electron is listed in contributions, Sections 3.3.1, 3.3.2, 3.3.3, 3.3.4, 3.3.5 and 3.3.6.
- The PT interconnect connects Kernels as well as other parts of the system that belong to the PT. This interconnect is based on 2D Mesh NoC.
- Kernels are processors that execute the main application offloaded code. Each Kernel is made from an interconnecting structure (Electron interconnect is listed in contributions, Sections 3.3.4 and 3.3.7), a controller (processor), code memory and a number of Processing Elements (algorithmic primitives such multipliers or adders).
- Processing Tiles and Kernels offer Coarse grained, Fine grained and Instruction Level Parallelism.

4.1.2 Software

A software layer residing on the Host system is required to communicate with the coprocessor and to support Virtualization tasks. For Virtualization purposes this software

layer would dedicate Processing Tiles to different host applications or/and it would execute host applications until completion before switching in new applications.

The only Virtualization support provided by the Electron is to reset all data/state data structures to protect host applications running on the coprocessor from seeing data that belong to different applications.

4.2 System Design Flow

The system design flow diagram is shown in Figure 19. Some of the diagram steps are done in parallel. For example step 4 does not need to wait for the step 3 to be completed.

1. A C-compiler creates the application program DFG/CFGs or a pseudo code (explained in sections to follow).
2. The custom created program parses DFG/CFG/pseudo code estimating required resources (Processing Elements - PE), and possibilities for concurrent data processing (type of PEs, serialized or concurrent Kernel design methodologies). Here we look for the best template match versus partial reconfiguration time. A set of templates exists on an on-board Flash memory. Another set of templates exists as a library of templates on the host computer. Templates provide a Virtualised Hardware model.
3. If adding a new template is our choice, the FPGA is partially reconfigured from the Flash device.
4. DGF/CFG/pseudo code is compiled to FPGA controllers' control-code.
5. Control-code and Data is downloaded to FPGA followed by initiating the FPGA.
6. After the processing is completed and results made available to the host, processing of the new application code starts.

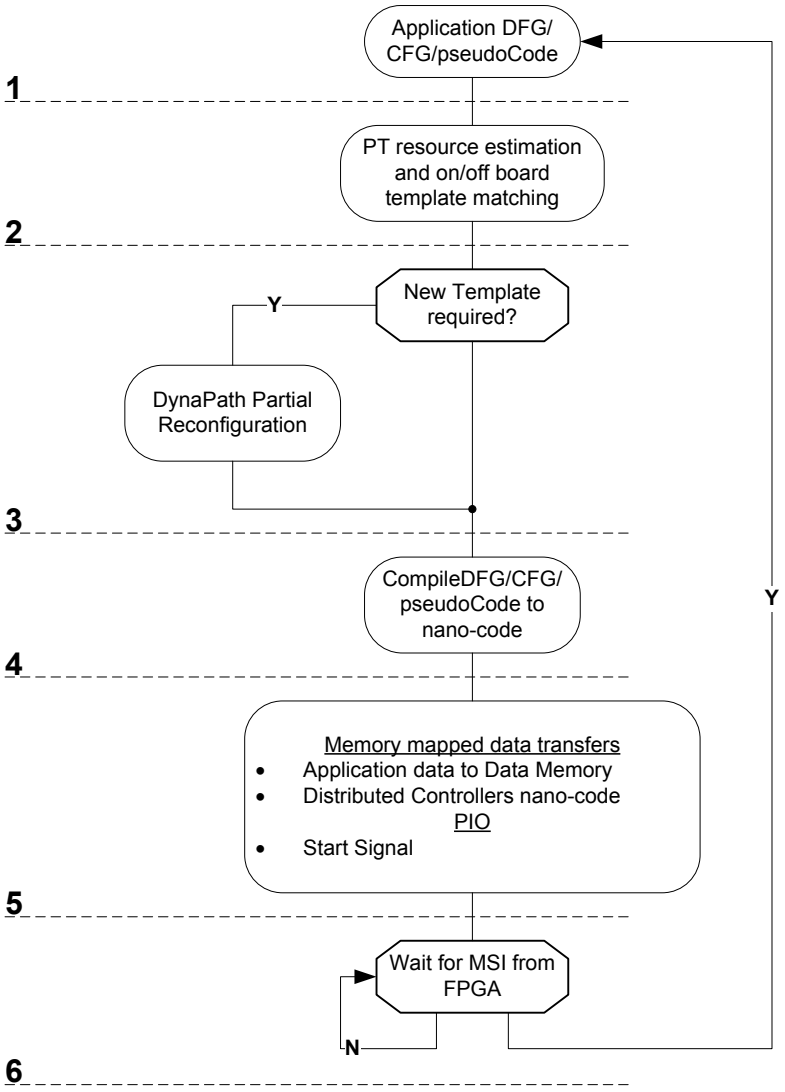

Figure 19 - System Flow Diagram

4.3 Processing Element Interconnect

In this section we analyse a Crossbar and 2D Mesh and how they can be used as the Processing Element interconnect. The goal is to build interconnect having a data flow processing capabilities.

4.3.1 Processing Element Crossbar Implementation

Fundamental concepts of crossbars are given in Section 2.4.1.1. This section gives Processing Element Crossbar (PXB) implementation details. The PXB is of a fundamental

importance. We need to determine what features it offers, its complexity and how to minimize complexity without affecting performance.

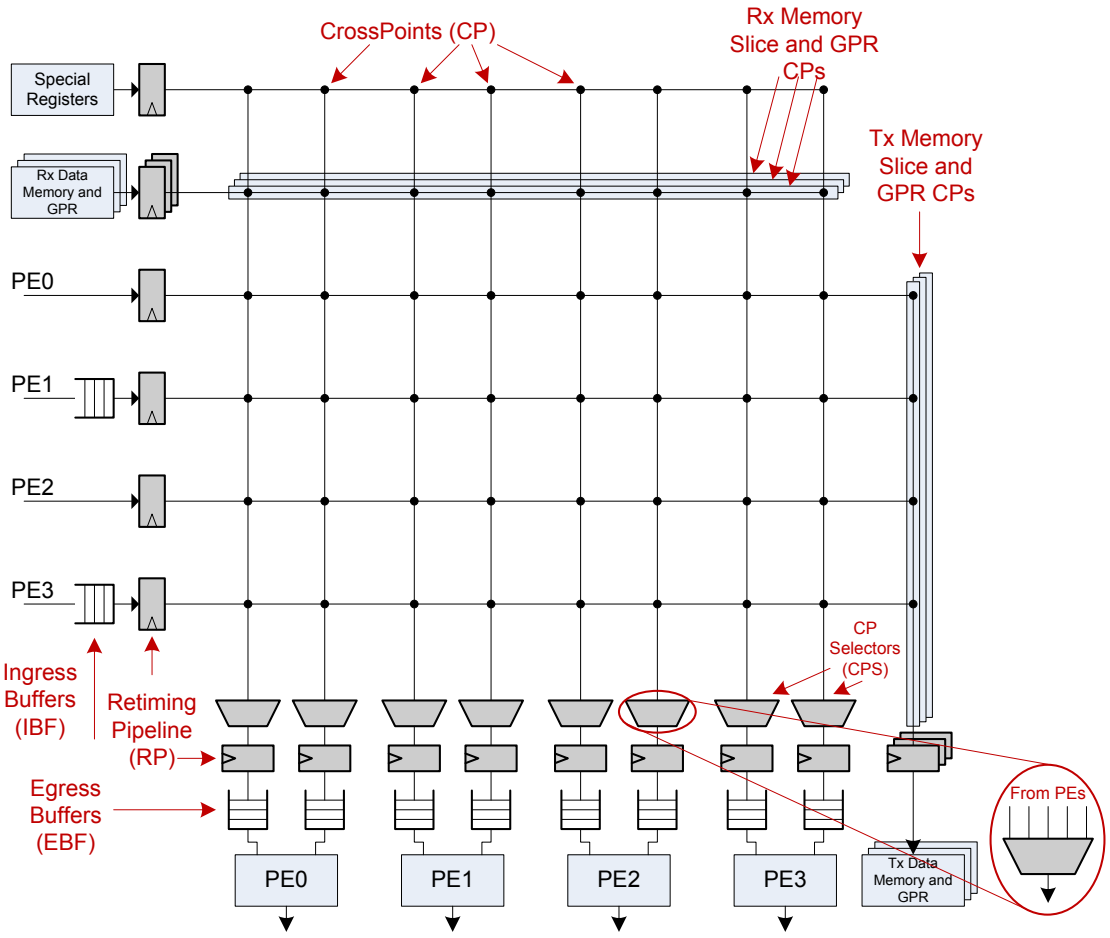

Figure 20 - Processing Element Crossbar (PXB)

4.3.1.1 Description

Figure 20 depicts a Processing Element Crossbar, description follows:

Outputs from all Processing Elements (PE) are connected to the left PXB side via optional Ingress Buffers (IBF) or directly to PXB input flops. This PXB side is referred to as the ingress side.

Inputs of all PEs are connected to PXB Egress Buffers (EBF). This PXB side is referred to as the egress side.

The Role of IBFs and EBFs is to help in input and output (resp.) clock cycle alignment as this may be required due to differences in PEs and PXB latencies. In some applications IBFs and EBFs could be used for clock domain crossing purposes.

Optional Retiming Pipelines (RP) are placed on ingress and egress sides of the PXB. The role of these pipelines is to improve the Physical Design (PD) allowing for faster clock rates. For some application, IBFs and RPs may converge and only RPs or IBFs could be used. A similar comment could be made for RPs and EBFs.

Crossbar Cross Points (CPs) represent connections between the PXB ingress ports (source) and the PXB egress side ports (destinations).

For example, Special Registers (SRs) have one output bus. This bus connects to N PEs via CPs. For every Processing Element SRs connect to only one PE input port. The CPs could be a set of wires, a set of flops or a FIFO.

Cross Point Selectors (CPS) are multiplexors placed on the PXB egress side. Their role is to select one of the possible CP busses connected to this PE. Figure 20 shows that the PE0 left input port has 6 CPs connected to its CPS. Therefore this CPS can select one of the 6 input busses.

In the practical part of this project all PEs are designed as fixed function components. All PEs have known and fixed latency therefore they lend themselves well to streaming architectures.

Receive data memory is a system level component used to store incoming data provided by the Host. To allow concurrent processing, Rx memory is designed from a number of slices. The number of slices and the data width is determined based on a number of criteria: PXB data width, number of PEs, the Host system bus (PCIe) width etc. Having more slices helps processing concurrency but increases the number of PXB ports therefore PXB complexity.

Transmit (Tx) memory is a system level component used to store results.

The Processing Element Crossbar contains all IBFs/EBFs, RPs, CPs and crossbar wires. It is desirable to minimize the number of the crossbar ports or Cross Points to minimize FPGA/ASIC resource and area usage.

Special registers (SR) can be used to store miscellaneous constants, results of compare operations or the status of IO operations (pass, fail, data integrity errors). These values could alter program execution. For example if the status value is FAIL, then abort further processing. Special Registers are accessible to write to or read from in parallel – this constraint is much different from GPRs[17]. It is expected that there are more GPRs than SRs therefore requiring the GPR data structure to be built from SRAMs. SRAMs have limitations such as the number of SRAM ports and the SRAM port contentions.
The following is an example of Special Register content:

SR[0] (ZERO), value == 32'd0

SR[1] (ONES), value == 32'hFFFF_FFFF

SR[2:15], registers storing miscellaneous values, results of compare operations, PE status, could be compile time defined.

4.3.1.2 Resources

Registers (flip-flops) and cross point selector are the major PXB components therefore we would like to know total resources taken.

For the above shown crossbar architecture:

$$FlopCount = ((Ports + Slices + 1) + ((Ports \times 2) + Slices)) \times BusWidth \quad [1]$$

Assuming 3 slice Data Memory and 32b wide ports:

$$FlopCount = ((4 + 3 + 1) + ((4 \times 2) + 3) \times 32 = 608 \quad [2]$$

Assuming the cross point selector (CPS) architecture defined in Section 4.3.1.4.2 we calculate number of CPS inputs:

[17] Not shown on the previous figure

$$CPSinputCount = numberOfCPs \times BusWidth \quad [3]$$

$$CPSinputCount = (Ports + Slices + 1) \times 32 = 8x\ 32 = 256 \quad [4]$$

Therefore each of the CPS has 256 inputs, grouped in 8 x 32. This number corresponds to 256 AND gates and 256 OR gates per one CPS. The above PXB has 8 CPS, therefore all CPS require 2000 (2 input) AND and 2000 (2 input) OR gates.

The AND or the OR gate have: ((2x numberInputs) +2) = 6 transistors, so total is 4000 x 6 transistors = 24k transistors.

Assuming that an average flip-flop consists of eight NOR gates, this corresponds to 8 x 4 = 32 transistors. Therefore 24k transistors correspond to 750 flops.

Summary: PXB requires approximately 1360 equivalent flops. As it is shown in Section 5.4.3 just one multiplier takes 679 flops. Therefore focusing design efforts on selecting PEs, their number, type and features is more important than minimising the crossbar. A similar conclusion was made for the coprocessor implemented in the practical part of this thesis, Section 5.4.4.

4.3.1.3 Multiply and Accumulate

As an example consider a DSP multiply and accumulate (MAC) reduction operation. MAC is a basis for a number of DSP algorithms, FIR and FFT[18] being some of them:

$$a \leftarrow a + (b \times c)$$

MAC function allows concurrency and streaming calculations. The following figure depicts a relevant PXB.

[18] Finite Impulse Responses and Fast Fourier Transformation resp.

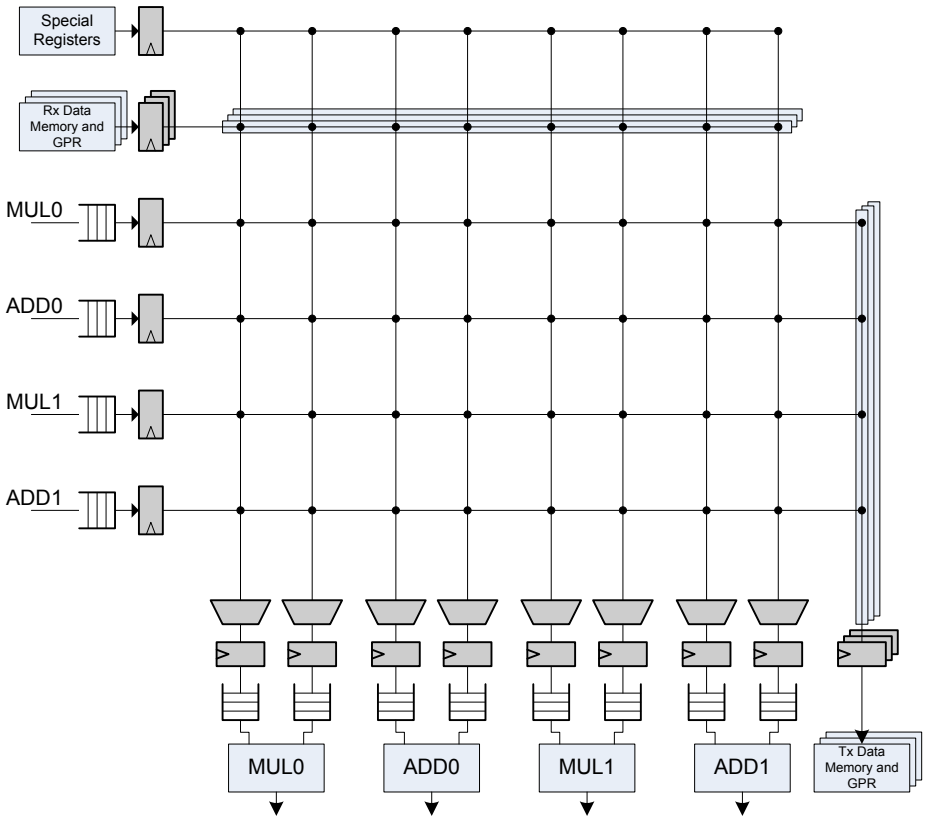

Figure 21 - Multiply and accumulate concurrent and streaming calculations

PXB contains two sets of multiply and adder PEs. The ADD PE has a feedback path on one of its input ports.

The following figure shows a sequence of streaming operations:

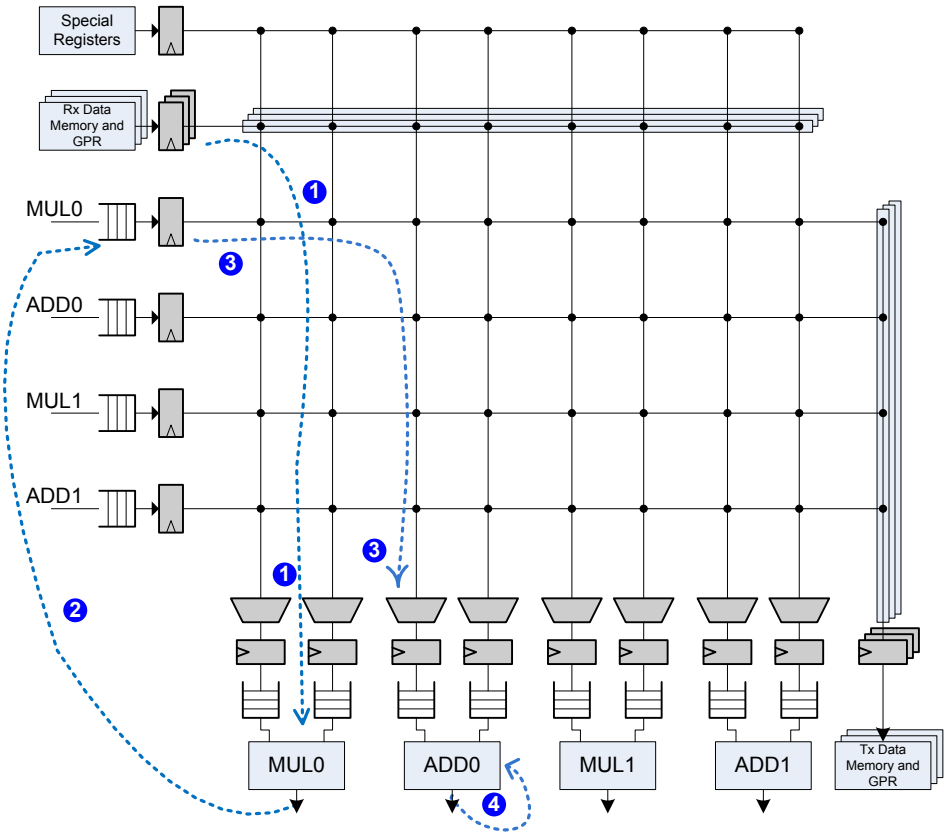

Figure 22 - Multiply and accumulate concurrent and streaming calculations

1. Processing data is read out from Rx Memory. Data progresses as a stream to both operands on MUL0 (therefore Rx Data Memory needs at least two slices). After a delay equal to RP (1 CC[19]), EBF (1 CC), multiplier primitive (the multiplier latency is Lm CCs) and output flop stage data makes it out from MUL0. The MUL is built as a pipelined multiplier therefore results (after this initial delay) are available on every clock.
2. MUL0 output data makes it through MUL0 IBF, RP and to RXB.
3. MUL0 output data is routed to ADD0 input A. The input B has a 0 value.
4. ADD0 output is looped back to the ADD0 B input port. Therefore ADD0 works as an accumulator.

The following time diagram gives all relevant CC:

[19] CC = Clock Cycle

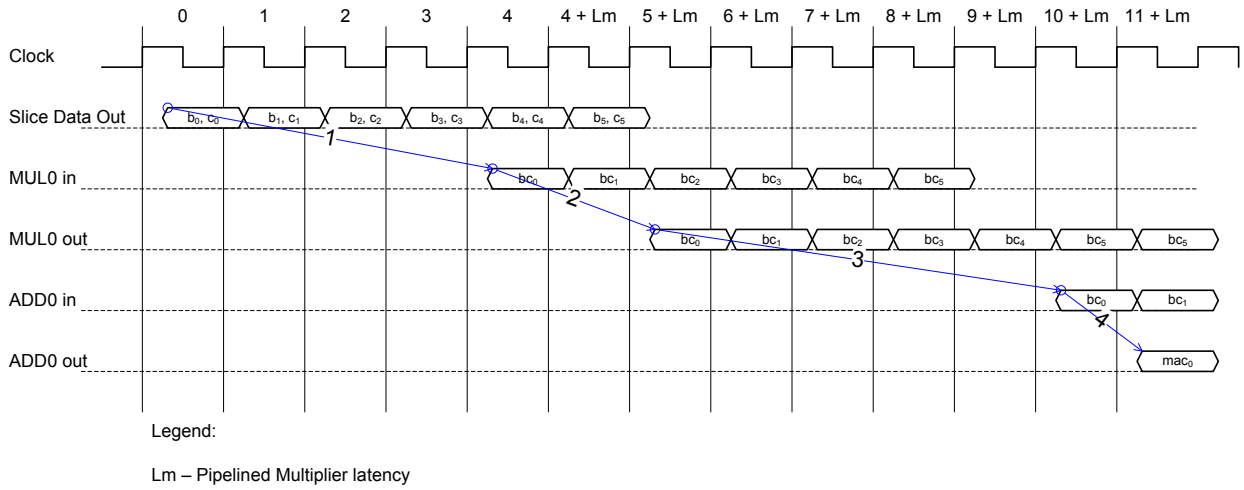

Figure 23 - Multiply and accumulate concurrent and streaming calculations - timing

1. In this timing diagram, Slice Data Out is processing data output by two Rx Data Memory slices.
2. Slice processing data is pushed into PXB. After PXB interconnect delay equal to 3 CCs (RP, RP and EBF) data is available at MUL0 input ports in a form of B and C operands (b_n,c_n).
3. The multiplier primitive block is designed as a pipeline. For assumed MUL latency of Lm CCs, after (Lm+1[20]) CCs data is at the MUL output stage.
4. MUL0 output data is switched through PXB (via IBF, RP, RP and EBF) and connected to ADD0.
5. For assumed ADD latency of 1 CC, the 1st MAC data phase is available at the ADD0 output (mac_0).

Therefore it takes Lm + 11 CC for the first MAC data phase to become available.

The above described process shows only one processing stream. It can be shown that a MAC operation can be divided into more than one concurrent flow effectively creating Single Instruction Multiple Data (SIMD) processing flow. Each flow is processing different

[20] There is a pipeline stage after multiplier primitive within MUL module

sections of the data stream. After all data stream sections are done their respective results need to be added.

Figure 23 shows the second processing stream running in parallel to the one described above. Processing data is read out from a different Rx Memory slice.

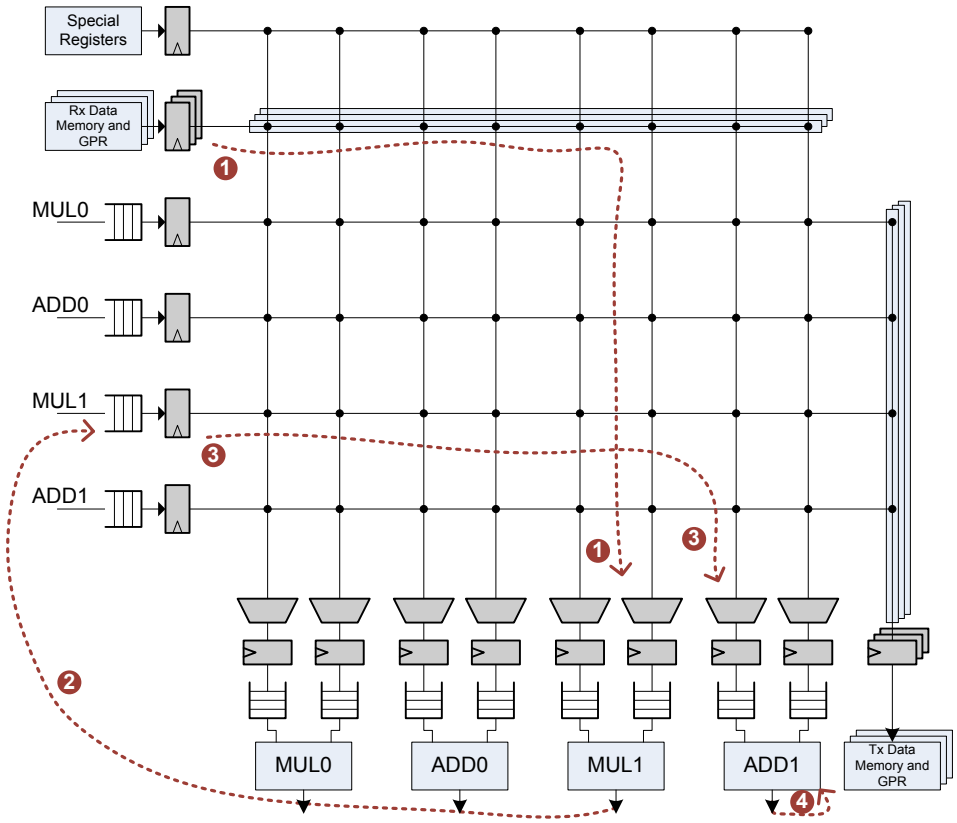

Figure 24 - Multiply and accumulate concurrent and streaming calculations

MAC results from these two streams need to be combined into the final results, Figure 25:

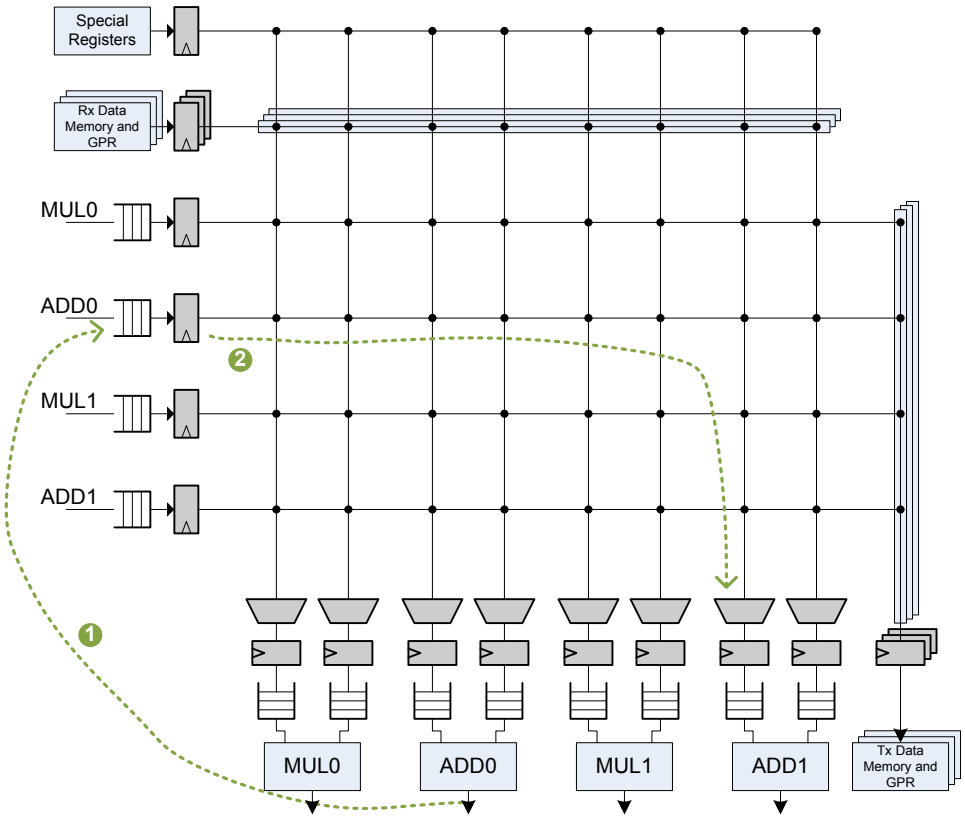

Figure 25 - Multiply and accumulate concurrent and streaming calculations

The final MAC result is written to Tx Data Memory, Figure 26:

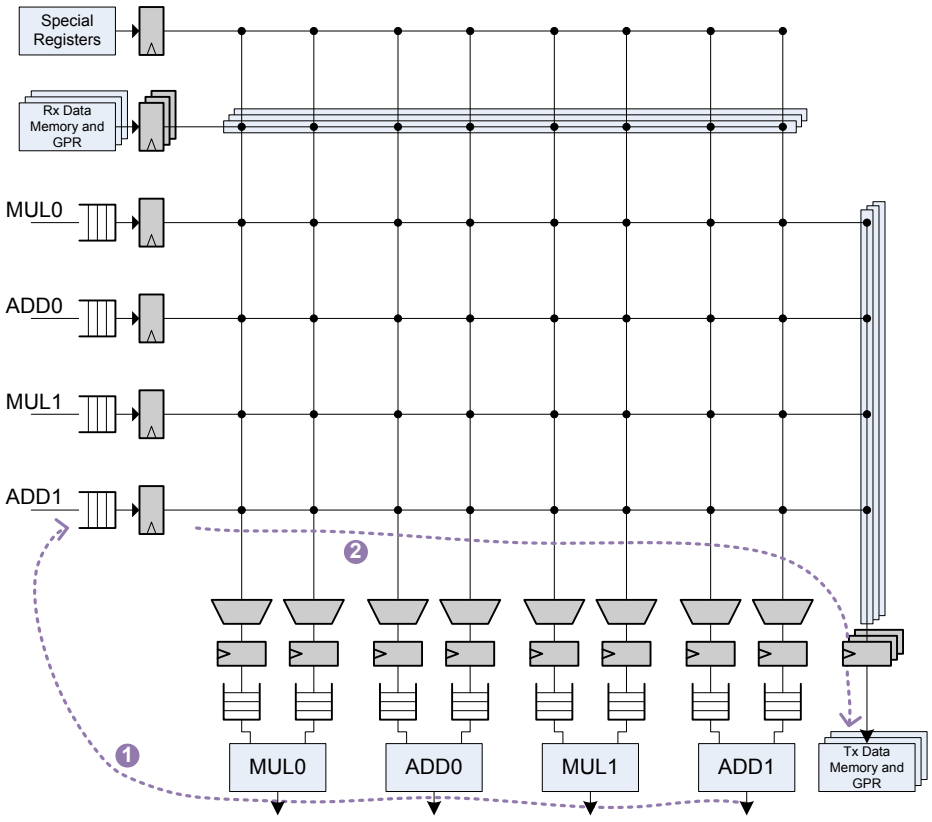

Figure 26 - Multiply and accumulate concurrent and streaming calculations

An equivalent circuit for the above example is shown on Figure 27:

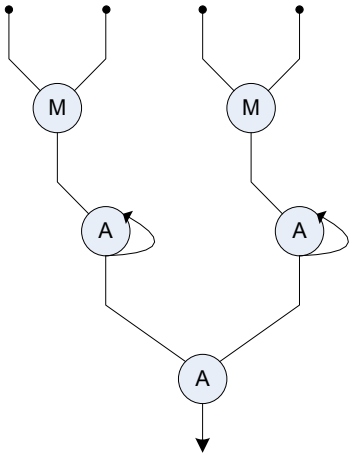

Figure 27 - Equivalent Circuit for PXB MAC example

In the following analysis we assumed that the multiplier and the adder are pipelined and can process a new operands one every clock (multithreading=1). In this case the MAC calculation time can be expressed by the following equation:

$$T_{mac} = (L_{rsxb} + L_m + L_{pxb} + L_{add} + \frac{dataset}{concurrency}) + (L_{pxb} + L_{add}) + L_{tsxb} \quad [5]$$

Legend:

T_{mac} - total number of clock cycles [CC] required

L_{rsxb} - receive memory slice latency through PXB

L_m – Multiplier MUL latency

L_{pxb} – PE PXB latency

L_{add} – Adder ADD latency

dataset – processing data set size

concurrency – number of concurrent calculations

L_{tsxb} – transmit memory slice latency through PXB

Assuming a dataset of 5000, L_m=5, L_{add}=1 and the above analysed MAC PXB configuration:

$$T_{mac} = (3 + (5+1) + 4 + (1+1) + \frac{5000}{2}) + (4 + (1+1)) + 3 \quad [6]$$

$$T_{mac} = T_{calculation} + T_{overhead} \quad [7]$$

$$T_{mac} = 2500 + 24 \quad [8]$$

It can be seen that:

1. Larger the data set is PXB interconnect and PE latencies are less important
2. More parallel processing is available the PXB interconnect and PE latency is more noticeable.
3. PXB interconnect and PE latency affects calculation time at the beginning (initial) and at the end (settling) of the streaming data processing.

The above summary shown as a generic delay formula:

$$T_{calculation} = L_{initial} + \frac{dataset}{concurency} + L_{settling} \quad [9]$$

At this point it is worth noting that a simple modification could improve how MAC processing would take place. Multiplier data rather than being routed to the adder via PXB would be delivered with a direct and local connection MUL to ADD PEs. However this modification would only shorten initial latencies, meaning for a large set of data this modification would not create significant benefits. This case is shown on Figure 28 with red arrows:

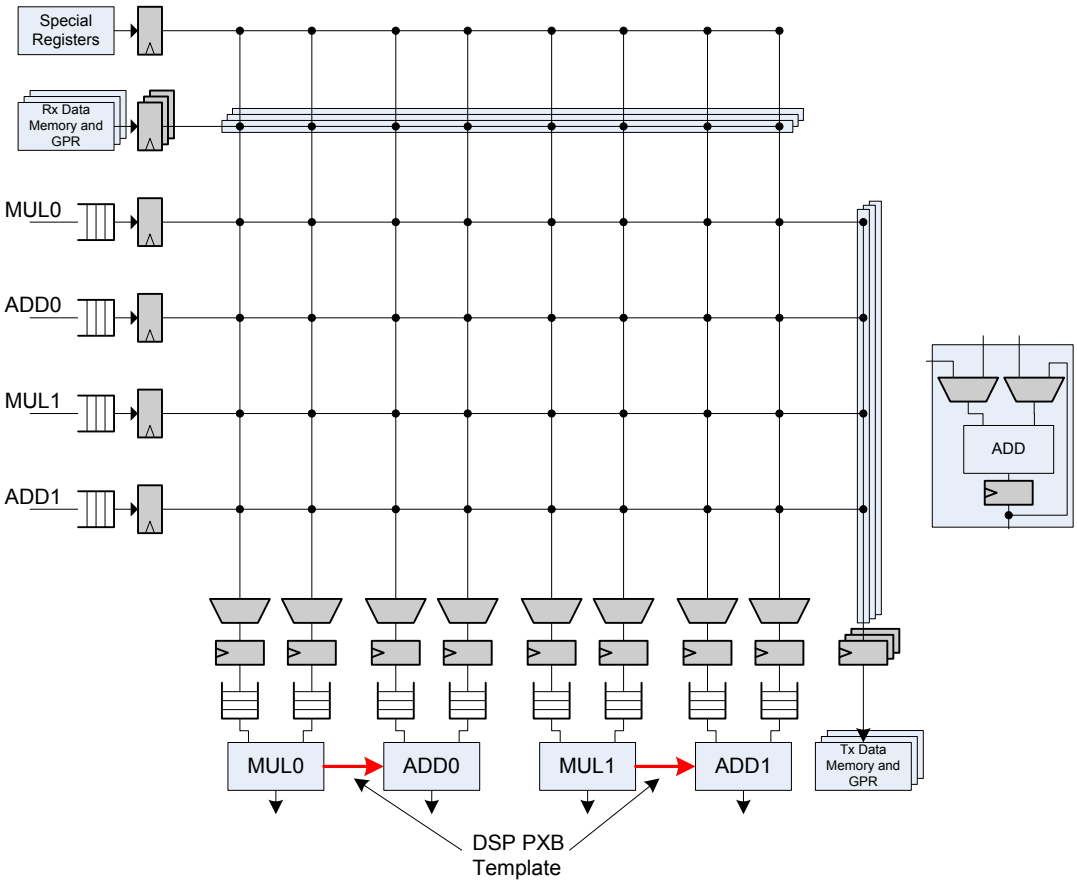

Figure 28 - PXB example for multiply and accumulate concurrent and streaming calculations

Section 4.6 extends this section by giving an analysis on multithreading effects on the processing speed.

4.3.1.4 Optimizations

4.3.1.4.1 Sparse Crossbar

It was previously indicated that minimizing number of PXB ports would reduce PXB resources. However that may not always be possible, ability to accelerate many applications may be lost.

The following example in Figure 29 shows how to reduce PXB resources by eventually reducing bandwidth for some applications.

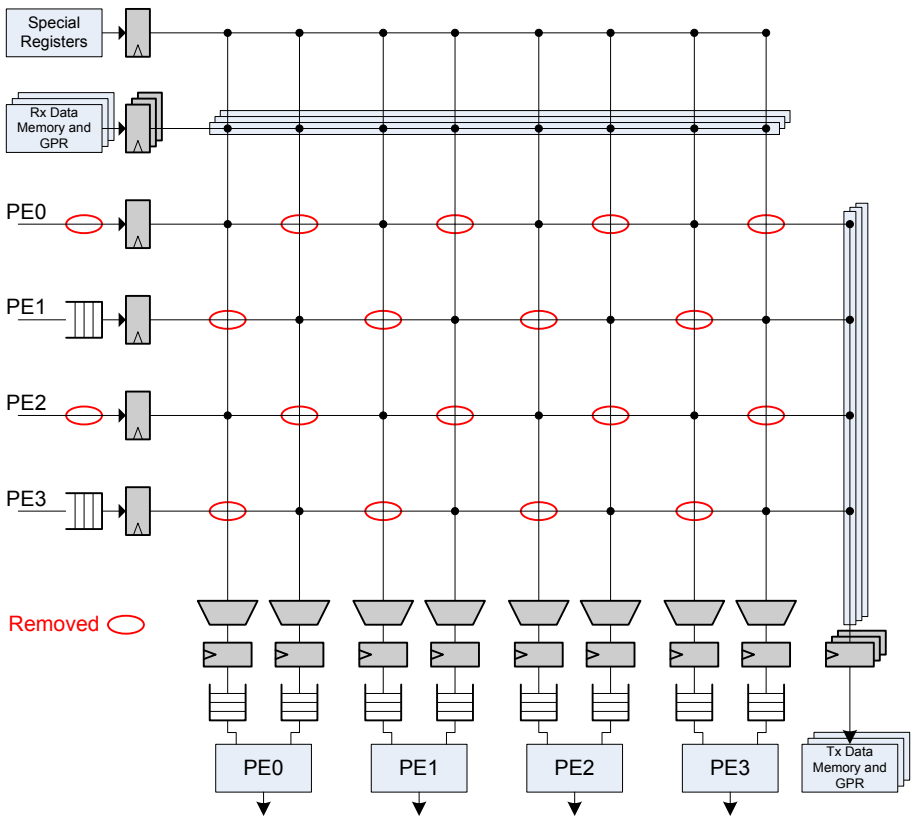

Figure 29 – Resource Optimized PXB – Sparse PXB

- CP – It is unlikely that a PE would use both operands from the same PXB port. For example PE_n may use one operand from PE_m but unlikely it would use both – therefore one CP can be removed. This CP optimization reduces number of wires on CPS to close to ½.
- CP – An optimization that does not compromise PXB performance at all is removal of SR to Tx Data Memory and Rx to Tx Data Memory CPs. This optimization is already implemented.
- EBF – These buffers may help in providing extra operation scheduling flexibility. However it may be less required to have them on long latency PEs. It is expected that EFB would be used on short latency PEs.
- Number of Rx/Tx Data Slices could be reduced from N down to 1 without loss of performance for application that would require serial processing of data.

4.3.1.4.2 Crossbar Selectors

Crossbar selectors (multiplexers) need to select one input from a number of inputs. Their design is important to avoid FPGA backend timing issues (signal propagation delay exceeding the clock period).

The following selector circuit is connected to one-hot schedulers, therefore depicted selectors lead to the lowest resource usage.

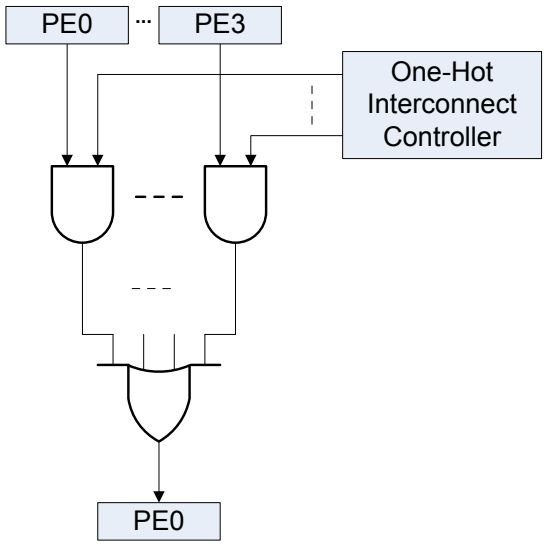

Figure 30 - Crossbar Input Multiplexing

4.3.1.5 Control

In this section we determine a number of signals required to control N x N PXB. A non-optimized PXB is assumed as shown on Figure 21 and for an optimized PXB refer to Figure 29. In both crossbars pass-through CPs are used (wires only).

PXB channels are defined by ingress to egress connections; there is no control signals required for each of CPs. On ingress side "valid data" signals are required. On egress side an input is selected by controlling egress side multiplexers (CPSes).

Non-optimized PXB, Figure 20, ingress side signals:

Special registers: N *5 - assuming 32 special registers

PEs: N * (1b write to buffer + 1b read from buffer) = N * 2b

Rx Data Memory Slices: N * 16b – assuming 64k address space.

Non-optimized PXB egress side signals:

PEs: N * ((N for multiplexor one-hot select + 1b write to buffer + 1b read from buffer) = N * (N + 2b)

Tx Data Memory Slices: N * 16b

Therefore total number of signals:

$$signalsTotal = N\ x\ (5 + 2 + 16 + (N + 2) + 16)$$

We find the optimized PXB, Figure 29, control signals following the above process. For the optimized PXB we assume:

- All RX and all Tx Data Slices use the same address bus,
- PEs connect to N/2 PEs,
- Selection of PE connected to egress port is done by decoded bus, rather than one hot bus.

$$signalsTotal = ceiling(N/2,1) * ((5 + 2) + (ceiling(log2(ceiling(N/2,1)),1) + 2))$$

Table 2 and Figure 31 give required number of control signals for non-optimized (a) and optimized (b) PXB with N = [3..16]. N is driven by the number of processing elements attached to this interconnect. It is very unlikely N would be more than 8. We use N maximum value of 16 to show the analysis trend.

Most, if not all, of PXB variants would have number of control signals in between non-optimized and optimized versions. For example some application may want to use more than one address buses and pass complexity of packing processing data to the compiler.

Table 2 - Number of PXB Control Signals

N	Non-optimized (a)			Optimized (b)		
	Ctrl Signals	Memory control signals (address	Signals Total (a)	Ctrl Signals	Memory control signals (address	Signals Total (b)
3	36	96	132	20	32	52
4	52	128	180	20	32	52
5	70	160	230	33	32	65
6	90	192	282	33	32	65
7	112	224	336	44	32	76
8	136	256	392	44	32	76
9	162	288	450	60	32	92
10	190	320	510	60	32	92
11	220	352	572	72	32	104
12	252	384	636	72	32	104
13	286	416	702	84	32	116
14	322	448	770	84	32	116
15	360	480	840	96	32	128
16	400	512	912	96	32	128

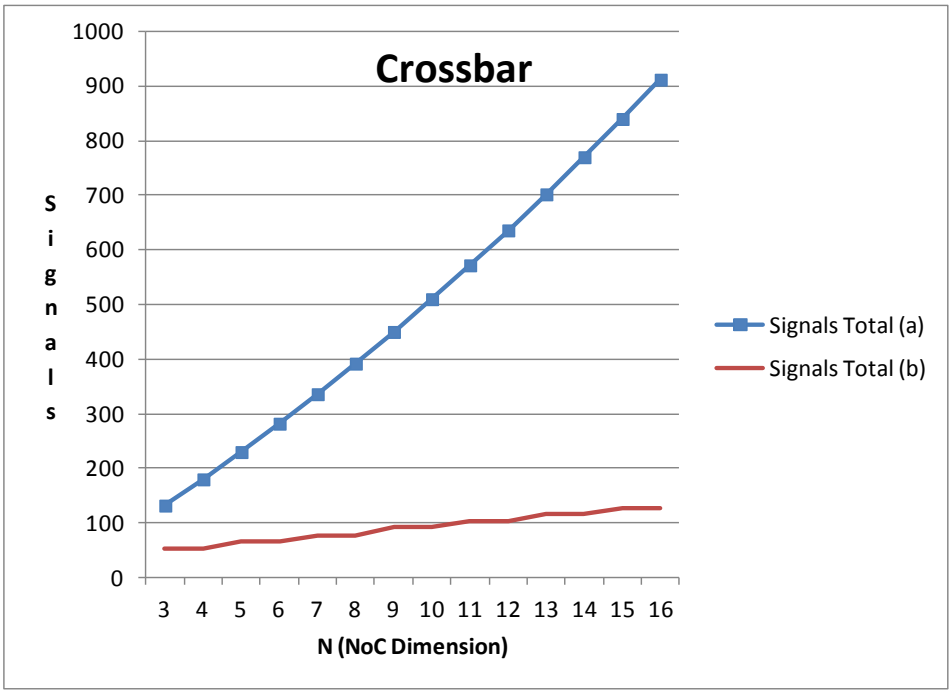

Figure 31 – PXB Number of Control Signals

Therefore we conclude that careful analysis of what are required PXB connections may lead into significant PXB resource savings. In the above case a resource optimized PXB has significantly reduced interconnect resources while keeping its flexibility for most of applications almost intact.

Section 5.2.3 gives an example of VLCW used in the practical part of this project. Presented VLCW is only 57-bits wide.

4.3.2 Processing Element 2D-Mesh Implementation

In this section we look into using a 2D-Mesh [90][91] (P2DM – Processing Element 2D-mesh). Fundamental concepts of 2D-Mesh are given in Section 2.4.1.2. Processing example used is a matrix multiplication. Processing elements are coarsened containing a multiplier and an adder together providing multiply and accumulate functionality (MAC) frequently used in DSP applications.

Coarsening processing elements brings a huge advantage of minimising traffic within any crossbar. At the same time we are running into risks of having PE that are not generic.

4.3.2.1 Array Multiplication

Here we are considering multiplying matrices A and B with matrix C as a result. Matrix size is p x p and relevant equations are:

```
c11  = a11 b11 + a12 b21 + a13 b31 + … + a1p bp1

c12  = a11 b12 + a12 b22 + a13 b32 + … + a1p bp2

c13  = a11 b13 + a12 b23 + a13 b33 + … + a1p bp3

c14  = a11 b14 + a12 b24 + a13 b34 + … + a1p bp4

…

c1p  = a11 b1p + a12 b2p + a13 b3p + … + a1p bpp

…

c21  = a21 b11 + a22 b21 + a23 b31 + … + a2p bp1
```

…

In a generic form:

```
cij  = ai1 b1j + ai2 b2j + ai3 b3j + … + aip bpj
```

Next we describe a simple yet effective algorithm that allows streaming matrix multiplication.

One way to solve the above described equations in a dataflow fashion is to process consecutive C elements (C11, C12, C13 etc) concurrently. It is required to stream all relevant A and B data and calculate C elements. Figure 32 shows an example where 4 C elements are calculated in parallel, starting from c11 (MAC 1), c12 (MAC 2), c13 (MAC 3) and c14 (MAC 4):

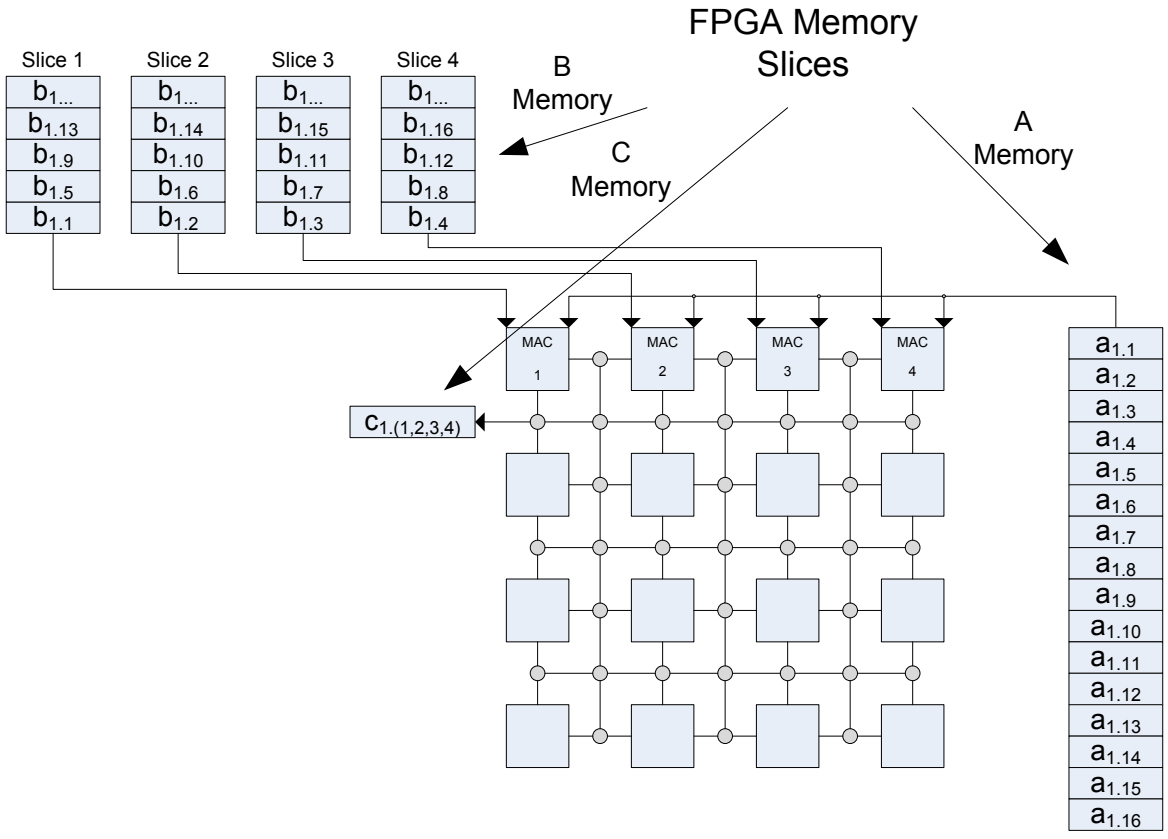

Figure 32 - P2DM and matrix multiplication example

In Figure 32 there are 4 MAC processing elements and 6 memory slices being used. The B memory is made from 4 memory slices each providing processing data concurrently. By using memory slices we provide required memory speedup for concurrent MAC operations. The A memory is made from one slice, therefore only one "a" element is available on any clock cycle. The C result memory is made from one slice and relies on the atomic writes of the 4 results (c11, c12, c13 and c14).

The following table describes matrix multiplication algorithm. Note that in any of shown processing cycles, n ϵ [1:p], the same A array element is used. This way we do not need as many memory slices for A as many we need for B. Value *ABpc* represents the previous cycle value that needs to be added to multiplication result from this cycle.

Table 3 - Matrix Multiplication

Clock Cycle / MAC	1	2	3	…	P
MAC 1 (c11)	a11 b11	a12 b21 + AB1pc	a13 b31 + AB1pc	…	a1p bp1 + AB1pc
MAC 2 (c12)	a11 b12	a12 b22 + AB2pc	a13 b32 + AB2pc	…	a1p bp2 + AB2pc
MAC 3 (c13)	a11 b13	a12 b23 + AB3pc	a13 b33 + AB3pc	…	a1p bp3 + AB3pc
MAC 4 (c14)	a11 b14	a12 b24 + AB4pc	a13 b34 + AB3pc	…	a1p bp4 + AB3pc

Memory slices are populated with data based on NoC configuration and number of relevant processing elements.

The above NoC configuration can be extended with the next NoC row with MACs 5, 6, 7 and 8. These MACs would need a new set of B slices. These new MACs would use A slice as the top four MACs. See Figure 33:

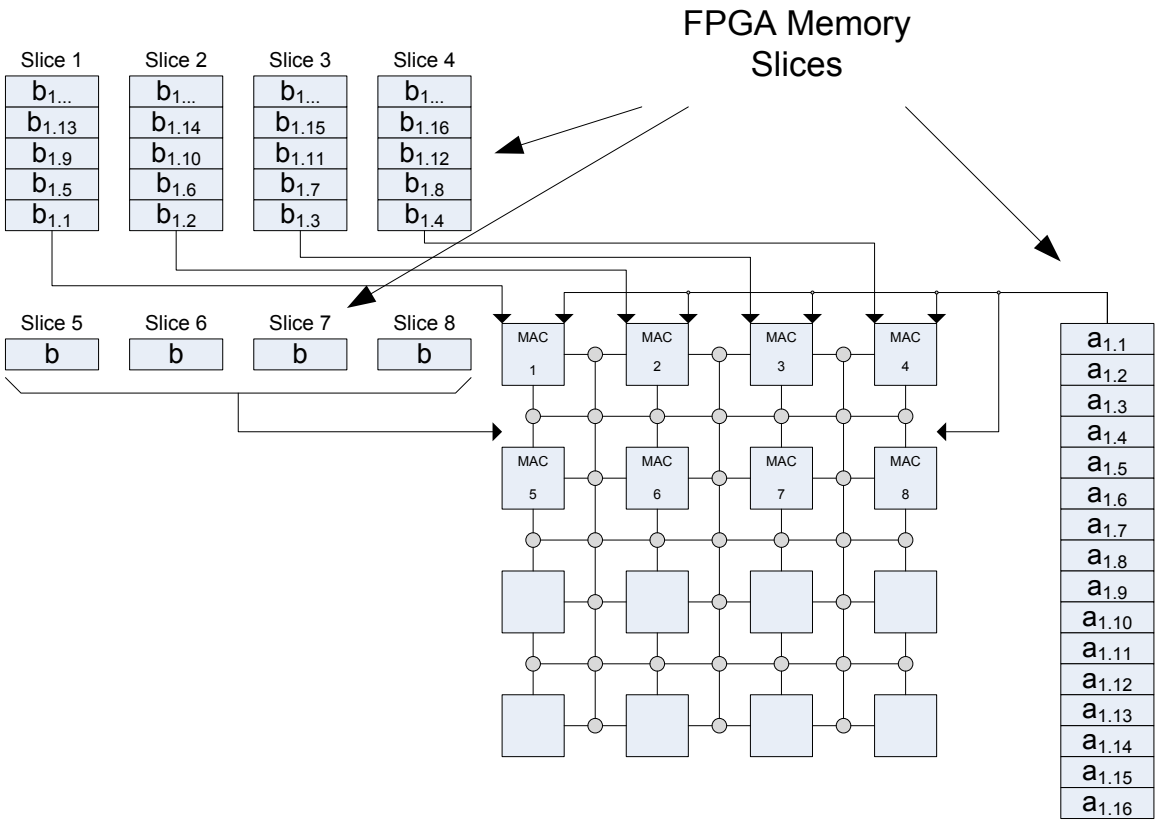

Figure 33 - P2DM and matrix multiplication example

This way of adding MACs and Slices would continue until all FPGA available resources are used. However there is a limit to slices since this number depends on number of BRAMs as well as how BRAMs can be overwritten with new A and B data.

4.3.2.2 Control

Here we determine a number of control signals required for non-optimized and optimized 2D-Mesh.

Non-optimized:

- All router ports can be accessed at the same time,
- From Figure 10 each of the cross point routers needs 3 control signals per port: two bits to define ingress ports (left, right or middle) and one signal to indicate if data is registered. Total 12 control signals per router.

- All memory slices feeding data to the same 2D Mesh row use the same address. Therefore there is N + 1 address bus (refer to Figure 33).

For the optimized architecture we assume:

- Router ports are programmed on back-to-back clocks
- CP routers contain register that keep interconnection. Therefore we need: enough signals to identify a router port ($\log_2$(totalPorts)), two control signals to define ingress ports and one signal to indicate if data is registered.
- All Memory Slices use the same address bus.

Table 4 and Figure 34 give required number of control signals for non-optimized (a) and optimized (b) interconnect with N = [3..16]. As it was indicated previously for PXB, N is driven by the number of processing elements attached to this interconnect. It is very unlikely N would be more than 8.

Table 4 – Number of P2DM Control Signals

		Non-optimized (a)			Optimized (b)		
N	CP Router Ports	Ctrl Signals	Memory control signals (address)	Signals Total (a)	Ctrl Signals	Memory control signals (address)	Signals Total (b)
3	56	168	64	232	9	32	41
4	120	360	80	440	10	32	42
5	208	624	96	720	11	32	43
6	320	960	112	1072	12	32	44
7	456	1368	128	1496	12	32	44
8	616	1848	144	1992	13	32	45
9	800	2400	160	2560	13	32	45
10	1008	3024	176	3200	13	32	45
11	1240	3720	192	3912	14	32	46
12	1496	4488	208	4696	14	32	46
13	1776	5328	224	5552	14	32	46
14	2080	6240	240	6480	15	32	47
15	2408	7224	256	7480	15	32	47
16	2760	8280	272	8552	15	32	47

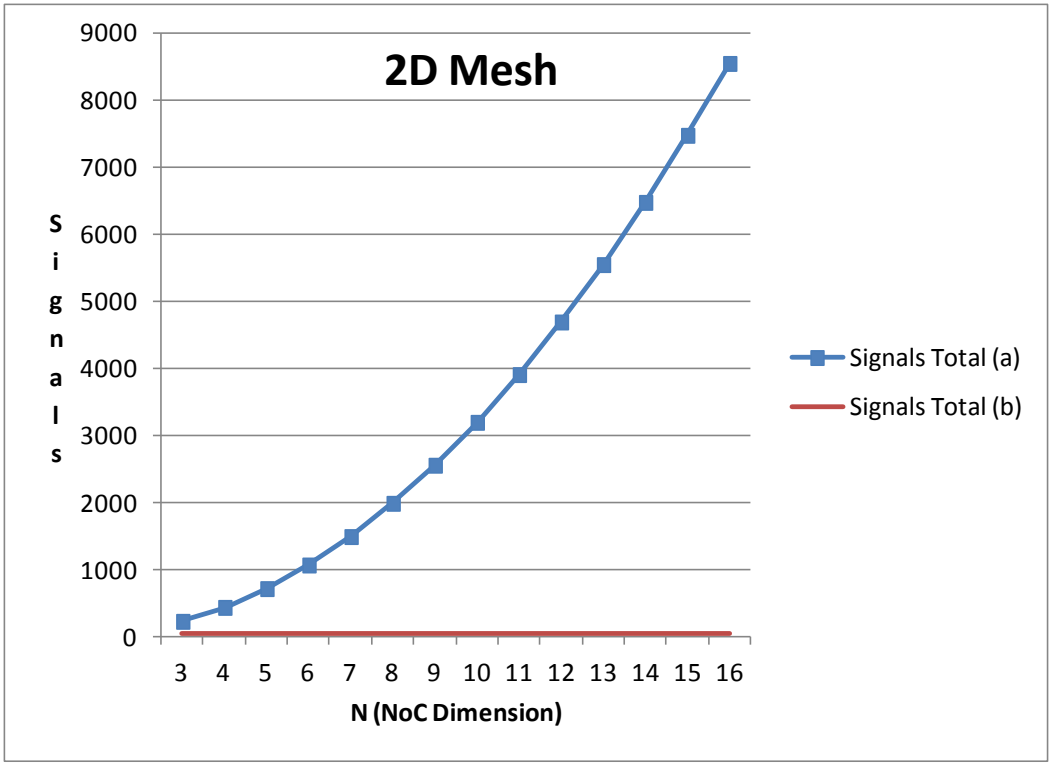

Figure 34 – 2D Mesh Number of Control Signals

Table 4, column "Non-optimized (a) Ctrl Signals" gives the number of router control signals if each router can be controlled and configured concurrently. The ability to concurrently change all router configurations comes with a cost of huge number of control signals, making such a 2D Mesh impractical for N larger than a few.

Table 4, column Optimized (b) Ctrl Signals gives the number of control signals if routers are configured at different clock cycles, therefore the control word that selects router ports can be encoded. However the Optimized 2D Mesh would require 1000s of clock cycles to configure all router ports.

Therefore a middle ground between practicality and performance is required.

4.3.3 Summary

In this section we analysed a Crossbar and 2D Mesh interconnect. Both interconnects allow data flow processing capabilities. 2D Mesh provides better features in terms of number of concurrent messages traveling through it. This is provided by its pipelined underlying structure. However it comes with an expense of a high number of control signals, making

the control word potentially too wide to be practical or if number of control signals is reduced, 2D Mesh may be to slow when its state is changed.

The Crossbar itself is smaller than 2D Mesh, requires less control signals, allows data flow processing and provides sufficient concurrent interconnectability for the application at hand.

Therefore in this thesis the 2D Mesh is used for a system level interconnect where a limited number of Kernels (Electrons) are instantiated, while the Crossbar is used as the processing element interconnect within an Electron.

4.4 Processing Tile

Processing Tile (PT) is a fully functional and self-contained multi-processor entity. In this project each Processing Tile hosts 4 Electron processing cores, called Kernels.

4.4.1 Control and Data Flow

From Figure 35:

- Processing Tile instantiates a System Controller. The System Controller is connected to all PT components via control buses.
- Each Kernel contains Electron Controller and a DynaPath (GPRs, SRs, PXB and PEs).
- Data Memory contains data that require processing. Data memory output (thick lines represent data buses) is connected to all Kernels via routers R1 and R2.
- Routers R1 and R2 allow data from the Data Memory as well as data from Kernel GPRs to be passed throughout the PT. PT interconnect is 2D Mesh.
- The final results is output from PT via the router R2 (arrow Result)

Processing Tile control and data flows are shown on Figure 35 and explained in the text that follows:

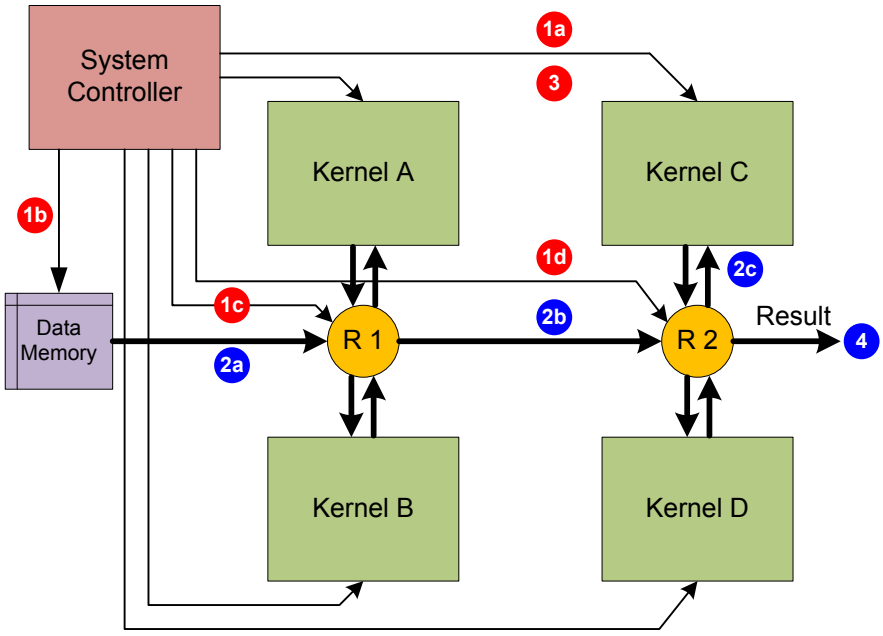

Figure 35 – Processing Tile Control and Data Flow

1. System Controller:
    a. Sends write strobe and GPR address to Kernel C.
    b. Sends read strobe and address to Data Memory
    c. Sends configuration signals to R1
    d. Sends configuration signals to R2
2. Data Memory:
    a. Outputs data based on System Controller provided read address
    b. Data from Data Memory makes it to R1 and from R1 to R2 and
    c. Data finally makes it to Kernel C where it is written to GPR address provided by System Controller
3. System Controller sends a start signal to Kernel C
4. Kernel C outputs Result to the Result FIFO, see Figure 49.

If Kernels work as a sequential pipeline, they store their respective results into their GPRs and System Controller reads result GPRs and forwards data to the next Kernels GPRs for further processing.

4.4.2 Memory Hierarchy

Processing Tile memory hierarchy follows Non-Uniform Memory Access (NUMA) model. Each Processing Tile Kernel has its own local memory (GPRs). GPRs are low latency accessible memories. Each Processing Tile has one instance of a shared memory (Data Memory) used by all Kernels.

Kernels can access data from a number of data structures. Access rate to different data structures varies:

- Kernels read input data from their respective GPRs and write intermediate results back to the same local GPRs.
- Kernels can indirectly access intermediate data from GPRs belonging to a different Kernel. This latency is larger than the latency to local GPRs.
- Kernels can indirectly access data from the shared memory. The shared memory access latency is larger than the local GPR or different GPR Kernel access.

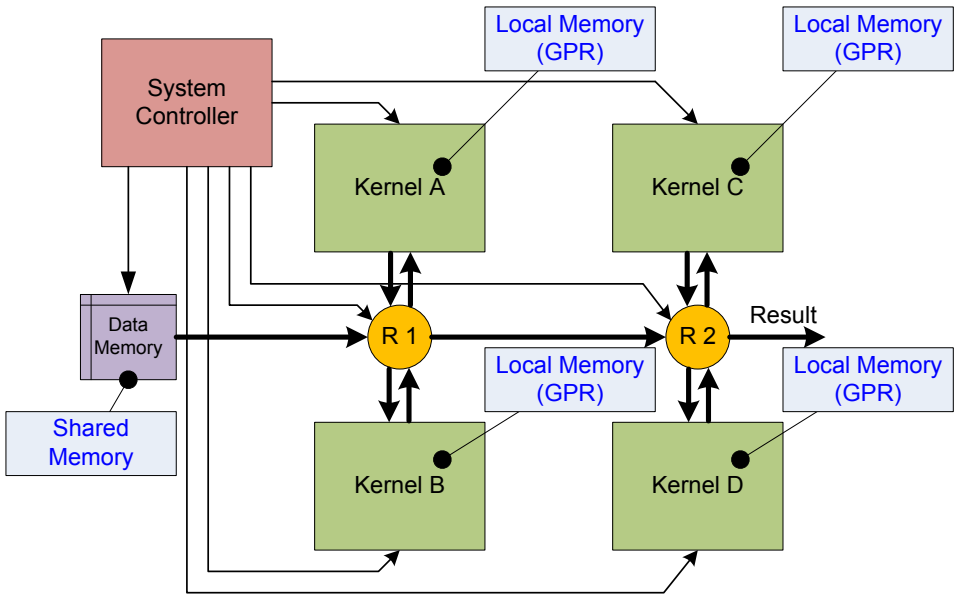

Figure 36 – Processing Tile NUMA

The System Controller takes the role of a memory controller. It reads the shared memory and writes data to Kernel GPRs. It also reads intermediate data from Kernels and writes this data to different Kernels.

4.4.3 System Controller

The System Controller, after being awakened by a higher entity, such as the Transfer Controller, generates all timing and scheduling event signals required to:

1. Control reading of Data Memory
2. Control Routers by connecting Router inputs and outputs
3. Transfer data from Data Memory to the Kernels General Purpose Registers (GPR)
4. Transfer intermediate data between Kernel GPRs and
5. Transfer final results from Kernel GPRs to the Processing Tile output (Result arrow)

The System Controller is based on Electron architecture; therefore it runs nano-code as well.

4.4.4 Data Memory

The Data Memory is built as 1R1W memory. The write port is used by a higher order entity such as the Transfer Controller. The read port is controlled by the nano-code running on the System Controller.

4.4.5 Router

The Router (Section 2.4.1.2) is a structure providing 4 input and 4 output ports. What input and output ports are connected is controlled by the nano-code running on the System Controller. The Routers are used to move processing data, intermediate data and final results throughout the Processing Tile.

4.4.6 Kernel

Kernels are instantiations of the Electron processor. Each Kernel executes part of a program and passes intermediate data to the next Kernel.

In this project each Processing Tile instantiates 4 Kernels. The goal is to map the Host application acceleration task into the 4 kernels for the best overall program execution. As it will be shown in Section 5.2.6 dividing the Host application acceleration task so each

Kernel executes its part of a program with similar time delay, provides the best overall performance (Section 4.8.2.2).

4.5 Electron

4.5.1 Electron Topology

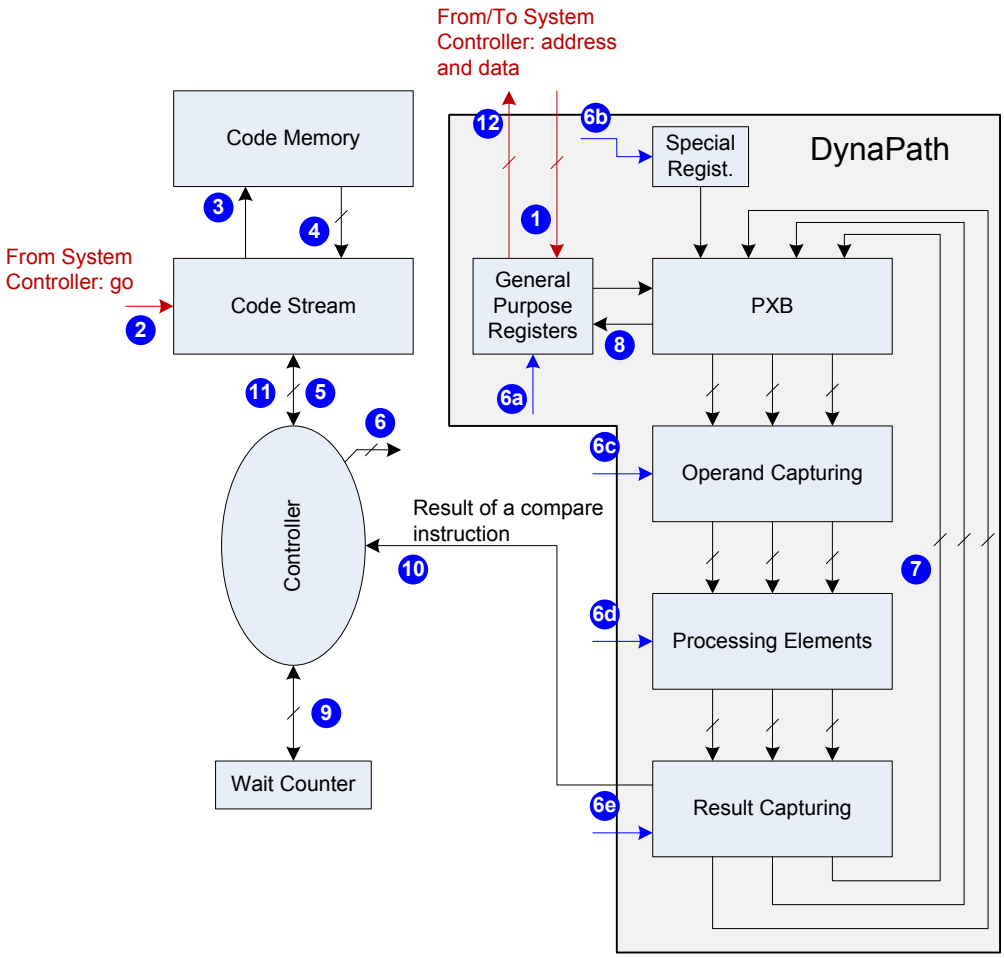

Figure 37 – Electron Topology

1. System Controller writes application data to GPRs. GPRs are organized as 1W1R memory. Both ports are shared by the System Controller and by Electron itself. This sharing presents no processing bandwidth issues since most of time only one of the two controllers need to access GPRs.

2. System Controller starts Code Stream controller

3. Code Stream controller starts reading Code Memory by providing an address.

4. Code Memory outputs instruction. Code Stream unit does partial instruction decoding looking for a program Branch instruction or program Done.

5. If instruction is not a branch instruction, Code Stream unit just forwards instructions to Electron control state machine.

6. Electron processor's state machine parses instructions and forwards clock cycle accurate control signals fanning them out to modules that require them.

6a. Electron creates an address and a read/write signal. If a GPR read was requested, the GPR data is output and placed on PXB. If GPR write was requested PXB data (intermediate data or the final result) is written to a GPR.

6b. Special Registers (SR) are accessed similar to GPRs but SRs can be accessed in parallel. Special registers store values often required for equation evaluation: 0, 1.0, -1.0, 0.5 etc. Properly specified Special register values can make existing processing elements more usable. For example multiplying a value by 0.5 would reply a divide by two etc.

6c. The Operand Capture stores operands required by Processing Elements. Operands could be values provided by Special Registers, GPRs or by Processing Elements via PXB. These values may have different arrival time therefore they may need to be stored until used. This module can be a FIFO or just one register flip-flop stage. In this project a flip-flop based capturing for all Processing Elements member inputs is used.

6d. Processing Elements is a pool of mathematical primitives or potentially more complex coprocessors. Operands required by Processing Elements are forwarded by Operand Capturing logic.

6e. The Result Capture stores results (final results or intermediate data) created by Processing Elements. Processing Elements results may have different arrival time therefore they need to be stored until used. This module can be a FIFO, as simple as a one entry deep FIFO or completely avoided. In this project a flop based capturing stage is implemented on all Processing Elements inputs.

7. Captured Processing Elements results are made available for further processing via PXB.

8. The final result(s) is written to GPR. The write operation is controlled by the Electron.

9. If an instruction requires Electron to wait (due to Processing Elements latencies) Electron loads a Wait Counter by the instruction provided value. This feature significantly reduces the code space and code memory size.

10. Electron may encounter a branch instruction (program "if" statement). Branch taken or not may depend on a value currently being evaluated by a Processing Element or on a GPR value. In any of these cases the branch taken or not is a result of some sort of compare operation. Compare operation operands could be intermediate data available from Processing Elements, Special Registers ("greater than 0" for example) or GPRs.

11. If instruction is a branch, Control Stream unit will forward the instruction and stop executing a program until Electron Controller instructs Code Stream controller if the branch is taken or not. If the branch is taken, Electron Controller also provides a new relative address. Code Stream adds the relative address to the current address to arrive to a new program address. This approach allows to multi-branch evaluation (not implemented in the current hardware).

12. The final result is read out by System Controller from Electron GPR.

4.5.2 DynaPath

DynaPath (Dynamically configurable Data Path) consists of the processing element crossbar (Sections 2.4.1), control signals, Special and General Purpose Registers and all Processing Elements attached to the crossbar.

DynaPath allows us to change how Electron data path is structured on clock to clock bases.

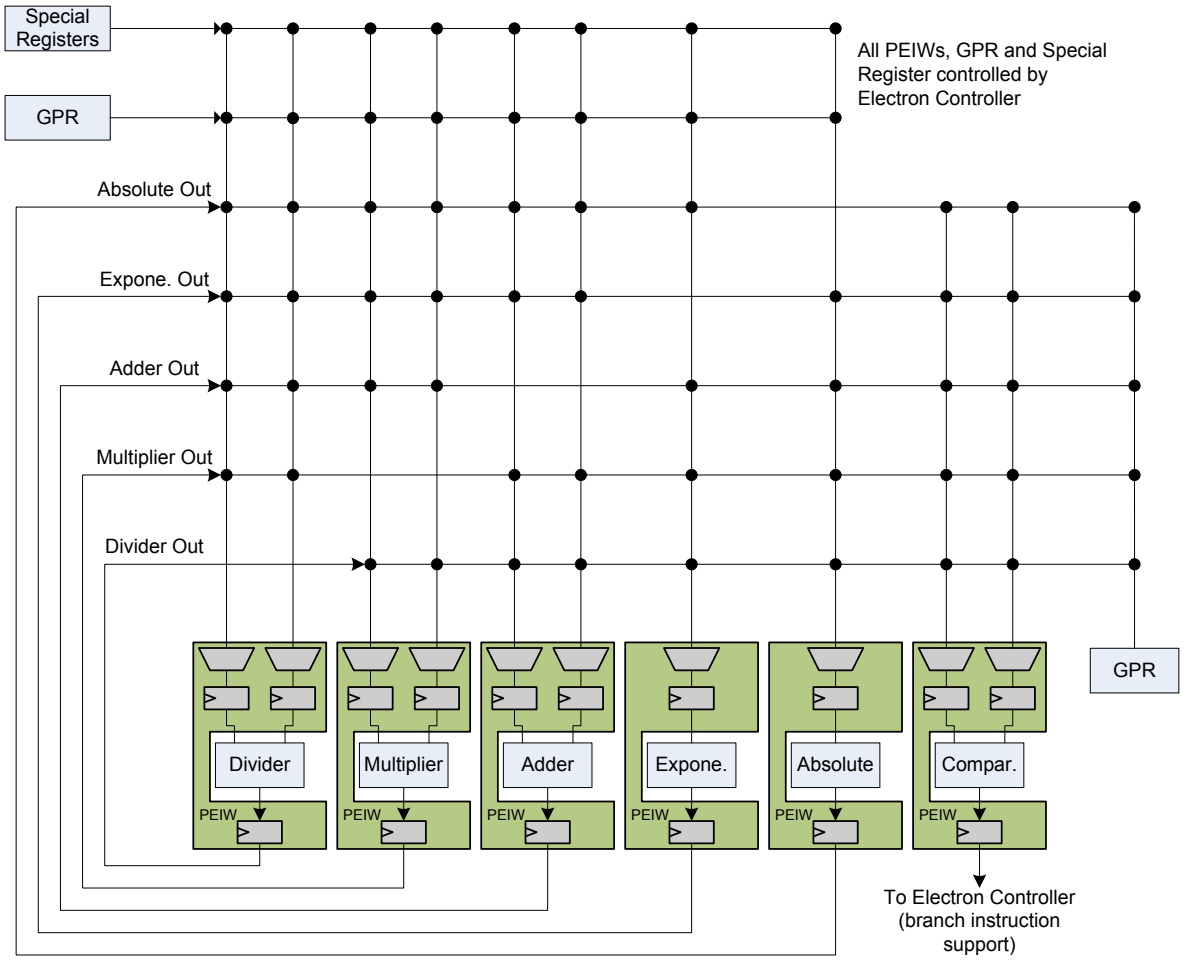

Figure 38 – Electron DynaPath

From Figure 38, General Purpose Registers (GPRs) are used to store processing data, intermediate data (intermediate level results) and final results. There is 16 32-bit GPRs. GPRs are designed as a register based memory with one Write and one Read ports. Both ports are shared with the System Controller. Since System and Electron controllers do not access GPRs at the same time port sharing presents no problem. The final results obtained by the Kernel are available to the System Controller via GPR read port.

Processing Element Interface Wrapper (PEIW) provides all PE interfacing services. It connects directly to the Crossbar. Under the Electron's direction the Crossbar input selectors located in PEIW select one of many crossbar outputs, independently for each of the two PEIW input ports. This independent input selection is used to deal with PE unbalanced latencies – we capture one operand until second operand is available. The

output flops are not strictly required. If used their purpose is to make timing closure easier. PEs can loop back their outputs to one of its inputs. The loopback functionality could be achieved via the Crossbar; however in this project we opted to use the sparse crossbar and PE internal loopback.

The compare PE is used to support a branch instruction. The compare output is sent back to the Electron Controller. The controller uses this signal to determine how to increment the program counter. The compare output is the only signal that goes outside of PXB.

4.6 Processing Elements

In order to increase the project productivity all algorithmic Processing Elements are created from Xilinx ISE Core Generator toolset.

Xilinx floating point primitives are used as part of hardware kernels called DynaPath. These primitives could be generated to offer:

- Pipelined processing where new operands could be processed on each clock and results output on back-to-back clocks after the initial latency. These primitives tend to be large.
- Non-pipelined primitives where new operands can be processed only once result of the previous operands is output. These primitives are smaller than their pipelined version[21]. Furthermore these primitives could come as a short latency version.
- Somewhere between pipelined and non-pipelined versions.

At the first sight use of fully pipelined primitives would allow building of very powerful processing kernels – but they would be large, therefore fewer kernels could be instantiated inside a FPGA. However, a case could be made that only a limited set of applications could take a full advantage of such primitives. Those applications would need to be highly regular in nature - such as multiply and accumulate.

[21] As per Xilinix datasheet a floating point primitive that can output new results on every second clock is two times smaller than the one that can output data on every clock.

As an example consider Figure 39.

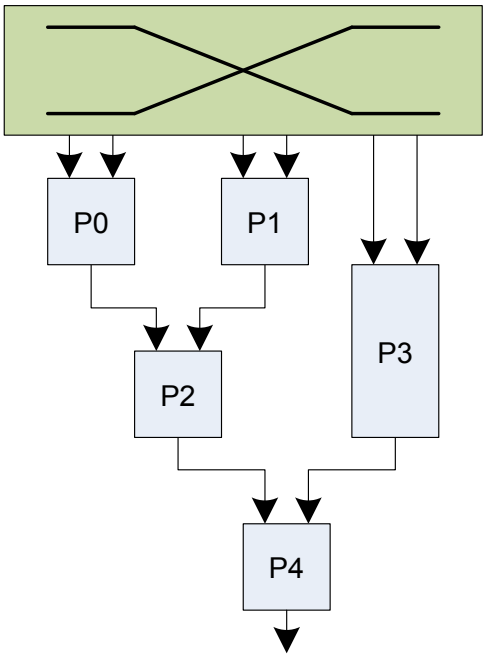

Figure 39 – Pipelined versus Non-Pipelined Primitives

From the figure, assume operation P3 takes place only once for the kernel's run-time. Its output is used by P4. P4 also uses output from P2. P2 is connected to P0 and P1 and uses P0 and P1 outputs as operands. Having P3 fully pipelined would be wasteful since this primitive is used only once.

As another example consider Figure 40. This time, outputs from P0 and P1 are used by P2 and P5 – but at different times. Therefore building P0 and P1 as pipelined primitives is beneficial.

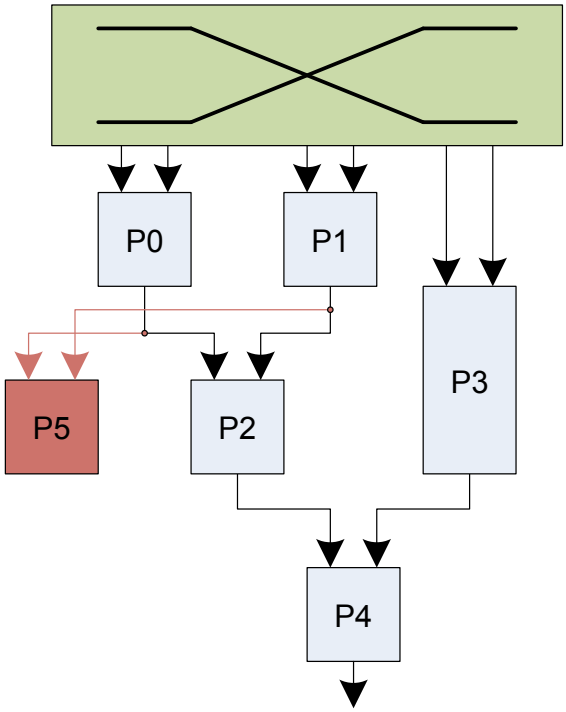

Figure 40 – Pipelined versus Non-Pipelined Primitives

For the purpose of this project a preliminary analysis of Black-Scholes formula revealed that due to sub-operation dependency there are no very many opportunities to use processing elements in a pipelined fashion. The only processing element that could take advantage of its pipelining version proved to be a multiplier. Therefore a design choice was made to have all processing elements except the multiplier as non-pipelined.

It is clearly understood that this generalization is not acceptable for all application and further study would need to be performed or a device could be built that would have some Processing Tiles built with primitives with different properties – therefore offering an optimum between logic size, flexibility and processing speed.

Next consider the MAC (multiply and accumulate) example provided in Section 4.3.1.3. There we analysed performance if the multiplier and the adder were pipelined with multithreading of 1 (new result available every consecutive clock cycle). Figure 41 displays how processing time is affected by the PE multithreading. As an example, if multithreading is four the multiplier can accept a new set of input operands every four clocks. In this case

the processing time is four times longer than if multithreading is one. It should be noted that multithreading affects latency, however in this example latency can be neglected (Section 4.3.1.3).

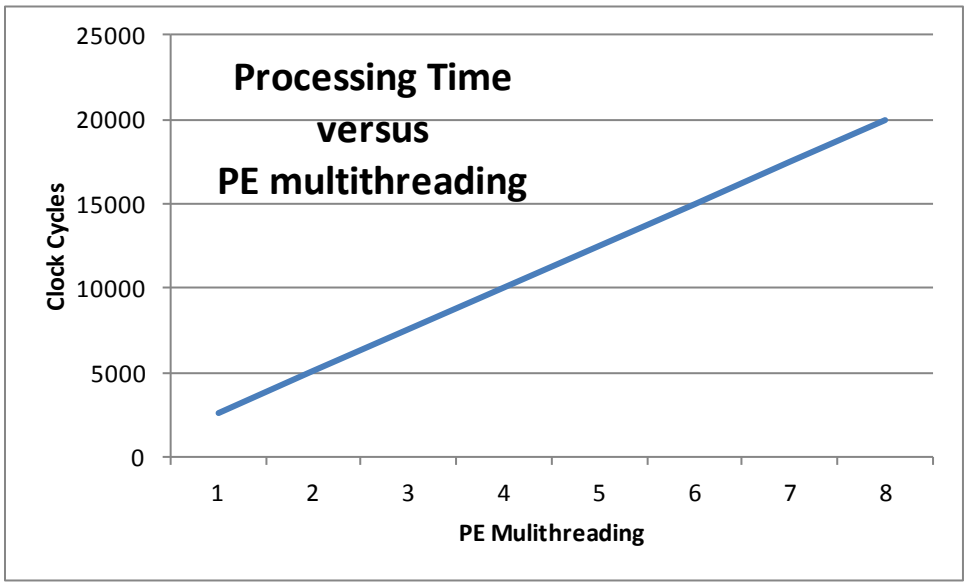

Figure 41 – Processing Time versus PE multithreading

Summary: The main point of this section is that the Processing Element selection and features are of high importance. They affect DynaPath size and consequently number of available Kernels (Section 5.4). They affect Kernel performance indirectly (size and number of available Kernels) and directly (multithreading effect).

4.7 Coprocessor Granularity and Parallelism

On the Coprocessor level, if all Processing Tiles and Kernels execute the same program, the complete Coprocessor works as a SIMD device.

The Processing Tile (PT) provides three levels of combined granularity and parallelism [106]. They are shown in Figure 42. All three levels are statically controlled by the nano-code compiler.

Coarse grained – the System Controller schedules all Kernels taking a role of a program outer "for" loop. The System Controller hosts required counters used by the "for" loop. In the practical part of this thesis, Chapter 5 -, System Controllers execute outer "for" loops.

Fine grained – processing Kernels provide Fine level parallelism. They can execute program inner "for" loops, smaller subroutines or subtasks. The system Controller controls and schedules Kernels. In the practical part of this thesis Kernels execute subtasks.

Instruction Level Parallelism (ILP) – every individual Kernel contains parallel data paths controlled by a VLCW control word. A number of instructions (arithmetic operations) can be executed in parallel. Besides ILP Kernels offer SIMD concept, see Section 4.3.1.3.

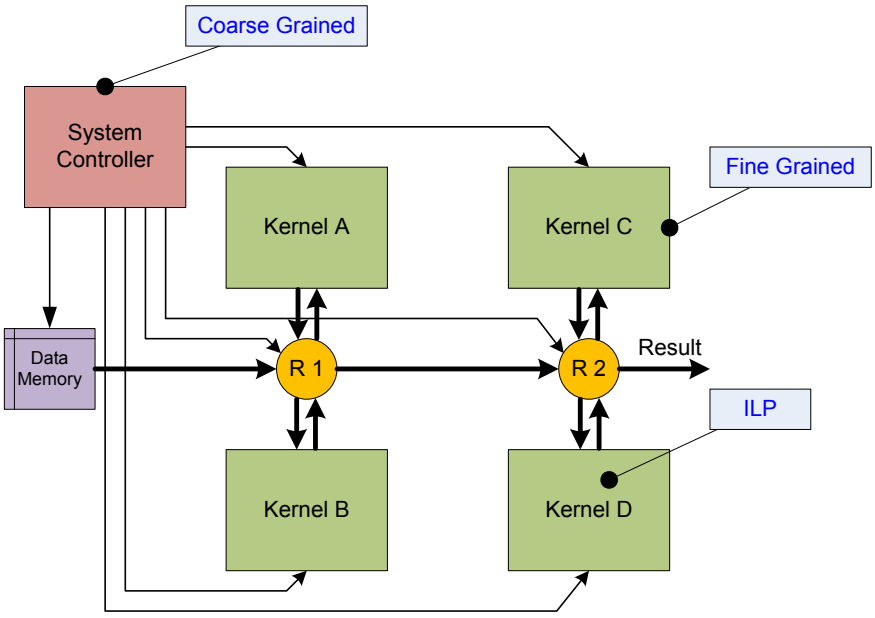

Figure 42 – Processing Tile Parallelism

4.8 Application Task Mapping to Kernels

In this section we look into mapping an application into Kernels. We are interested in performance obtained by two different FPGA configurations as well as how Computation to Communication ratio influences FPGA application task partitioning.

4.8.1 Serial and Parallel Scheduled Kernels

Section 2.7 deals with Amdahl's law. There it is indicated that the law sparked debates over multi-core processor architectures. Here we look into this particular Amdahl's law implication. In addition, we define two regions and analyse how Amdahl's law is applicable to a Coprocessor described in this thesis. The regions are defined by the relation of sequential and parallel applications sections.

4.8.1.1 Many versus few processing cores

We are interested in performance obtained by two different FPGA configurations. Both configurations are based on the same FPGA architecture having 6 Processing Tiles each having 4 Kernels. Configurations:

1. The application is divided over all available 24 FPGA Kernels where each Kernel runs the complete task until completion. We consider this as "many small cores" architecture.
2. The application is divided over all available FPGA Processing Tiles. Each Processing Tile runs the complete task until completion while Processing Tile Kernels run application sub-tasks. We consider this as "few powerful cores" architecture. For more information see Section 4.8.2.2.

We assume Black-Scholes computation. For the FPGA Configuration 1 the complete Black-Scholes computation is run by each Kernel. For the FPGA Configuration 2 see Figure 50 and Section 5.2.6. There it is shown that Processing Tile Kernel 1 takes ST1 (K1), Kernels 2/3 take ST2/ST3 (K2A/K2B) and Kernel 4 (K3) takes ST4. Kernels 2 and 3 execute in parallel.

We also assume many workloads are available (this is mandatory for CPU off-board Coprocessors, they are meant to do "for" loop processing) and that memory subsystem does not limit availability of new processing tasks.

Table 5 Show how Black-Scholes computation is divided into four subtasks (ST1 to ST4). ST2 and ST3 are the same in length, they perform the same processing. Next we consider three Scenarios (SC1 to SC3). SC1 described the case from the practical part of this thesis (see Chapter 5). In SC2, subtasks ST2 and ST3 are shortened from 136 to 90 cycles. In SC3 subtasks are rebalanced and of the same length.

Table 5 – Processing Subtasks in Clock Cycles (CC)

Subtask/ Scenario	ST1	ST2	ST3	ST4	Total [CC]
SC1	104	136	136	82	458
SC2	104	90	90	82	366
SC3	100	100	100	100	400

Figure 43 depicts FPGA Configuration 1 (bars PSC1, PSC2 and PSC3) and Configuration 2 (bars SSC1, SSC2 and SSC3) throughput results expressed in Million of Operation per Second, where an operation is one Black-Scholes evaluation.

Legend:

PSCn, P stands for Parallel, each Kernel processes application in parallel and until completion. SCn stands for Scenario one, two or three.

SSCn, S-stands for Serial, each Kernel from a Processing Tile processes the application subtask. SCn stands for Scenario one, two or three.

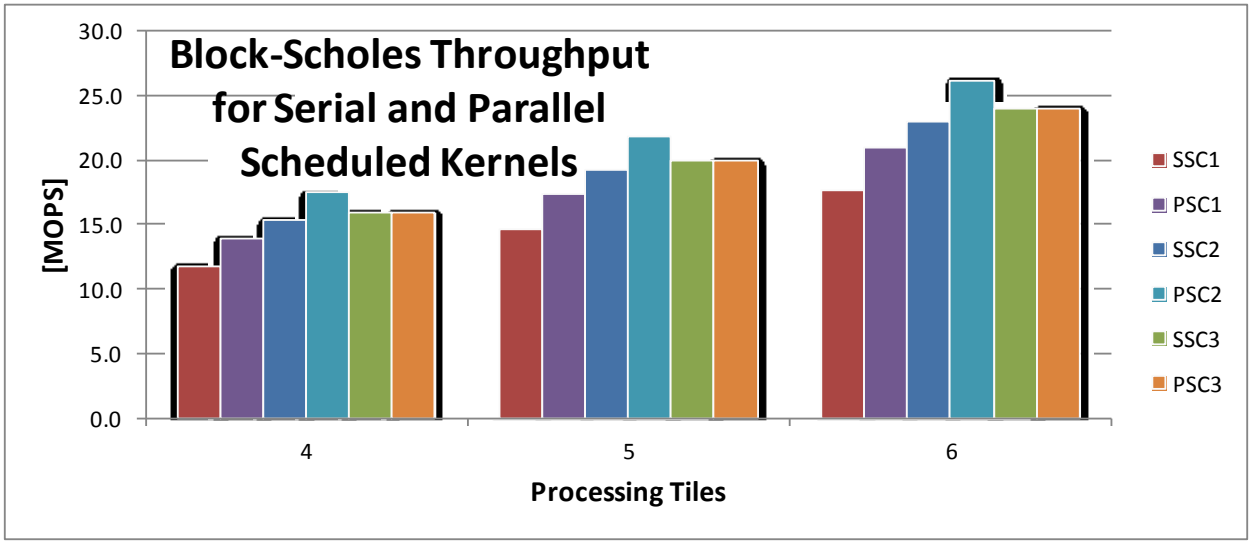

Figure 43 – Black-Scholes Throughput for 2 FPGA Configurations

From Figure 43 we conclude that regardless of the number of Kernels/Processing Tiles, Configuration 1 outperforms Configuration 2 in all cases but when subtasks are of the same length (cases PSC3 and SSC3). In this case both Configurations produce the same calculation throughput. The reason why Configuration 1 performs better is a consequence of how serial Kernels are scheduled in Configuration 2. The slowest Kernel determines the speed of the complete Processing Tile. A new processing task cannot be scheduled until the slowest Kernel is done; we have a Kernel run-time rounding effect. On the other hand Configuration 1 does not have this rounding effect, Kernels run exactly as long as required.

4.8.1.2 Amdahl's Law Applicability

For Amdahl's law applicability to an FPGA Coprocessor see Figure 44. This figure shows two cases:

Case 1 - parallel code section (Tp) is shorter than serial (Ts). In this case adding more parallel capabilities to reduce Tp would lead to no overall throughput improvement. The serial code section dictates when a new set of data can be scheduled in (see Sections 4.8.2.2 and 5.2.6).

Case 2 - parallel code section (Tp) is longer than serial (Ts). In this case adding more parallel capabilities to reduce Tp would lead to the overall throughput improvement.

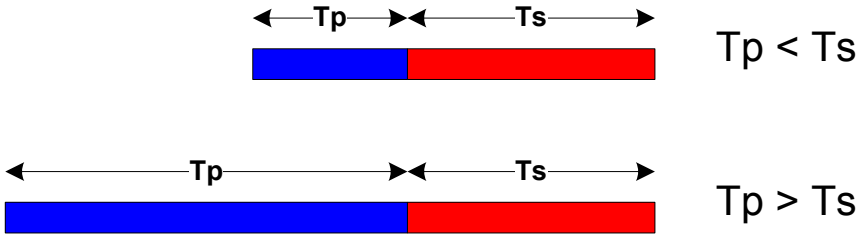

Figure 44 – Amdahl's Law applicable to FPGA Coprocessor

4.8.2 Computation to Communication Ratio

If a memory subsystem were 100% efficient it would provide new data on every clock. In this case the processing subsystem could be designed as a pure sequential pipeline. This pipeline would create a result on every clock.

However, most memory subsystems are not able to deliver data on every clock. A well planned design should have the memory subsystem and the processing logic speed match. If not matched:

- Processing is too slow – therefore memory subsystem will be flow controlled
- Processing is too fast – therefore we have excess of processing resources

In this section we examine "Computation to Communication" [6] ratio scenarios. First we analyse an application that allows parallel processing following with an application that requires sequential processing.

4.8.2.1 Parallel Processing

We show that increasing the computing processing power is beneficial to the application processing time to a point where the communication speed wall is reached. Further processing speed improvements can be made with increasing the computation workload by coarsening the processing kernels and organizing the memory subsystem to provide data so more processing can be done for the same amount of data.

In a spreadsheet that follows we use matrix multiplication[22] as an example although the complete analysis is generic. We consider FPGA resources such as the number of processing kernels and the clock rate (computation) versus the external memory bandwidth (communication).

As it will be shown at one point the communication speed limits the computation speed. This is the point when the communication infrastructure cannot deliver enough data to enable all FPGA resources for concurrent operations.

[22] Matrix multiplication allows concurrent processing

Table 6 - FPGA resources and Processing Speed for a Matrix Multiplication

Kernels	Theorethical Computation Time/BW			DDR3 Achivable Time [S]
	Clocks	Time [s]	[Maps]	
1	1.00E+12	5000.0	200.0	5000.0
2	5.00E+11	2500.0	400.0	2500.0
3	3.33E+11	1666.7	600.0	1666.7
4	2.50E+11	1250.0	800.0	1250.0
5	2.00E+11	1000.0	1000.0	1250.0
6	1.67E+11	833.3	1200.0	1250.0
7	1.43E+11	714.3	1400.0	1250.0
8	1.25E+11	625.0	1600.0	1250.0
9	1.11E+11	555.6	1800.0	1250.0
10	1.00E+11	500.0	2000.0	1250.0
11	9.09E+10	454.5	2200.0	1250.0
12	8.33E+10	416.7	2400.0	1250.0
13	7.69E+10	384.6	2600.0	1250.0
14	7.14E+10	357.1	2800.0	1250.0
15	6.67E+10	333.3	3000.0	1250.0
16	6.25E+10	312.5	3200.0	1250.0

DDR3 External Memory			
Devices	Width	[Mbps]	[Maps]
1	16	25600	800

Legend	
FPGA clock [nS]	5.00E-09
Matrix dimension	10000
Matrix elements[M]	100
Data Width	32
Maps	Mil. Access per Sec.

From the above spreadsheet:

- The following computation parameters are changed: number of available computation kernels and the system clock rate. With fixed communication speed (800MHz DDR3), we differentiate 2 major computation corner cases:
  (a) Slow processing speed: system clock 5nS (200 MHz). This case is shown in Table

6.

(b) Fast processing speed: system clock 2.5nS (400 MHz)

- For the given matrix size and the number of kernels we calculate *Theoretical Computation Time/BW*. The *[Maps]* column defines the required communication throughput (external memory access rate) so FPGA processing engine would not be stalled.
- For the external memory transfer speed defined in *DDR3 External Memory* we calculate external memory bandwidth in *[Maps].* DDR3 technology given overhead (refresh rate, bank open/close or tRC requirements) is not considered
- For the given matrix size, number of kernels and FPGA clock rate we calculate matrix multiplication *DDR3 Achievable Time* based on what the external memory can offer in terms of speed.
- For each of the 2 FPGA clock rate scenarios we calculate the *DDR3 Achievable Time*, the computation time where the communication speed limits the processing time. For 5 nS FPGA clock the external memory BW limits processing time to minimum of 1250 [S] if 4 or more kernels are used.

The following chart gives us a graphical version of the above defined spreadsheet.

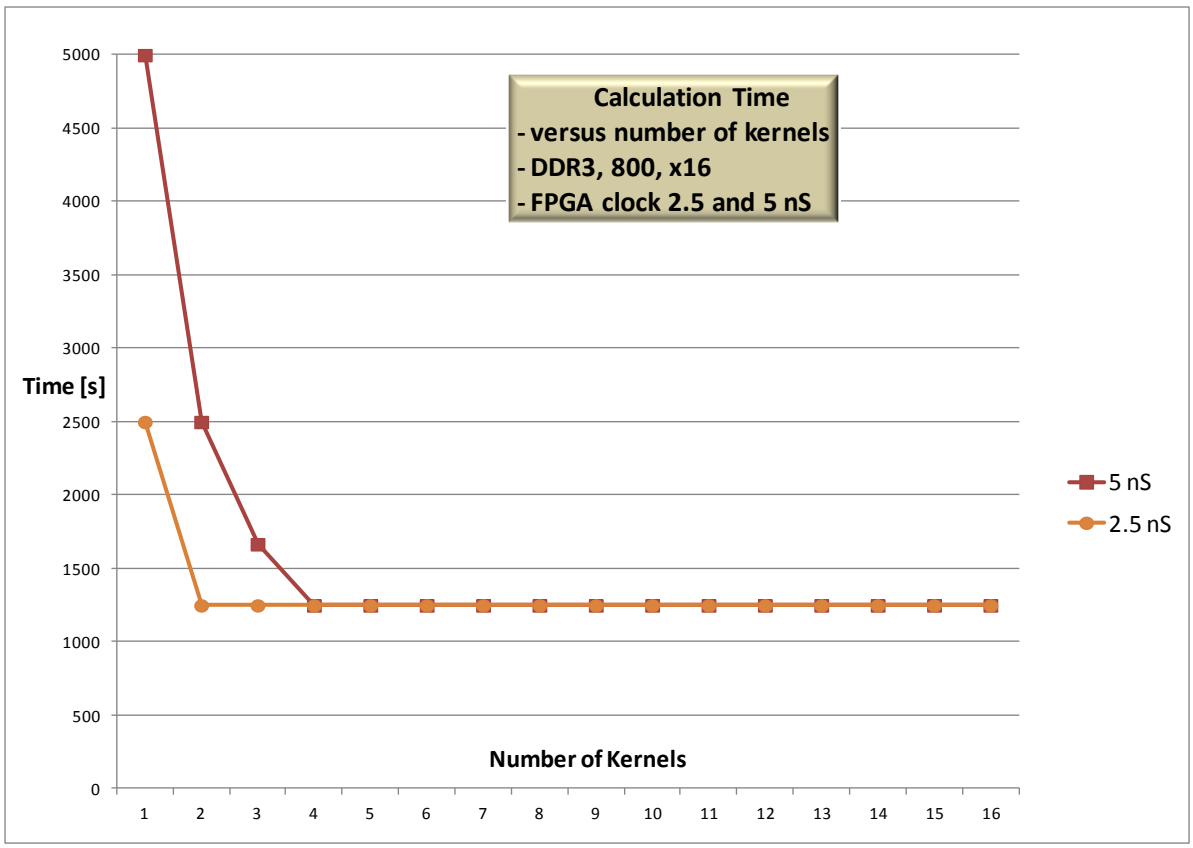

Figure 45 - 10k by 10k matrix multiplication time and external memory system bandwidth limitation

Our desire is to reduce the processing time as much as possible. Adding more computation power to FPGA will shorten this time however after one point increasing computation capabilities has diminishing or limited effects due to communication speed limits.

From the graph shown in Figure 45, going from 200MHz (5nS) up to 400MHz (2.5nS) the processing time got two times smaller or 50% decrease in processing time if the number of Kernels is [1..2] as it is expected by the clock rate change.

If the number of Kernels is 3, the 2 times faster clock gives only 25% processing time decrease. The further increase in number of Kernels does not yield to any improvements. The improvement in computation speed got limited by the communication speed.

To improve FPGA processing efficiency it is desirable that acceleration task has a few operations to perform[23] per each set of input data - therefore effect of the interconnect bottleneck would be minimized. In other words adding more workload to computation part of the "Computation to Communication equation" would have a positive effect of minimizing the acceleration system dependency to communication speed.

However this increase of computation workload may come with the cost of more complex algorithms. This cost is passed to the application parser and control code compiler. Their task is to optimize and structure input data as well as to group processing operations into kernels.

On the graph shown on Figure 46, it is assumed that Kernels for each set of data execute 1, 2, 4 and 8 operations - Kernel's workload is increased by two, four or eight times.

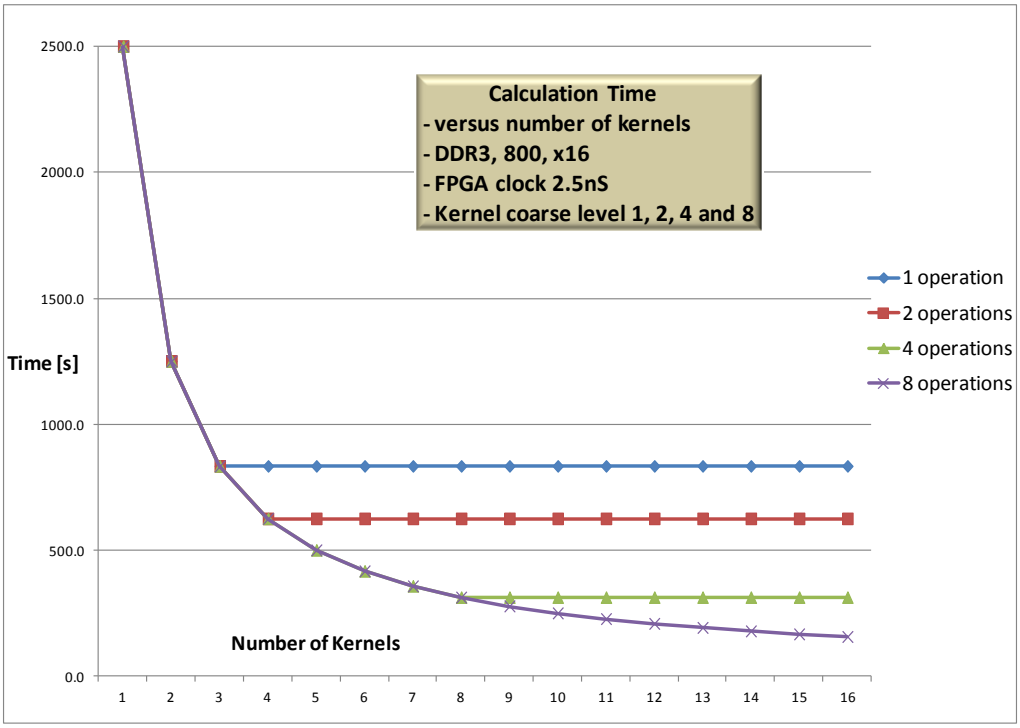

Figure 46 - Increasing Computation Workload Minimizes Communication Speed Wall Effect

[23] As an example, instead calculating a matrix multiplication followed by matrix inversion find an algorithm that can do matrix multiplication and inversion at the same time on a subset of data at a time

What this graph tells us is that increasing the kernel workload decreases the systems dependency to the communication speed wall.

Summary:

Computation time and communication speed are tightly coupled. Increasing computation capabilities above the kernel to kernel or kernel to memory communication bandwidth does not bring any advantages. However increasing computation workload for the same amount of processing data avoids the limiting effect of the communication speed wall. Increasing the workload may require use of enhanced software pre-processing algorithms and coarse grained processing elements (kernels).

4.8.2.2 Sequential Processing

Not all computation tasks can be divided into parallel sub-tasks. Some are sequential due to operation dependencies. Amdahl's Law tells us that this sequential task determines the application speedup factor. Some serial tasks can be divided into serial sub-tasks by creating a pipeline of processing units (Kernels), Figure 47. Each Kernel performs different serial sub-task and passes data and/or intermediate data to the next Kernel. The last Kernel outputs the final result.

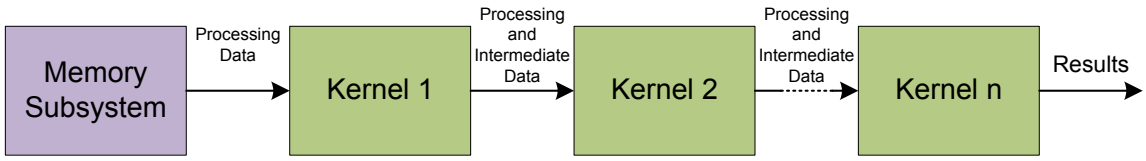

Figure 47 – Kernel Pipeline

The following figure depicts Kernel processing in time.

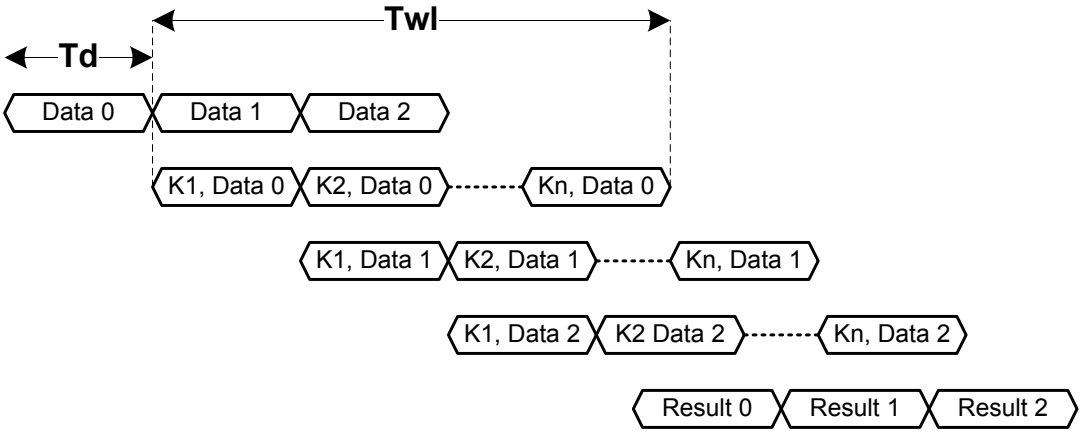

Figure 48 – Kernel Pipeline – Timing Diagram

Therefore the number of Kernels required:

$$N = \frac{Twl}{Td}$$

It is unlikely a processing task can be subdivided into N sub-tasks of the same duration. Two cases can be differentiated:

- The longest Kernel time is shorter or equal to Td. This case is not a problem. We allow each Kernel to run Td duration. The above figure depicts this case.
- The longest Kernel takes more time than Td. This case is a problem since now fast Kernels need to be scheduled to run as long as the longest Kernel. As a consequence we need to slow down reading from the Memory Subsystem. In effect we insert bubbles in the pipeline.

As a conclusion, we would like to divide the processing workload into Kernels of similar duration, where each Kernel individually has run time equal or less than Td.

For more analysis related to how many Kernels to instantiate refer to Section 5.3. There we use our experimental setup and analyse the maximum number of Kernels so our processing speed does not unnecessarily exceeds the input data bandwidth.

Chapter 5 - FPGA-based Coprocessor Implementation

5.1 FPGA Design

In this section we give a system level operation description. The system diagram shown in Figure 49 is based on the FPGA Coprocessor from Figure 18[24].

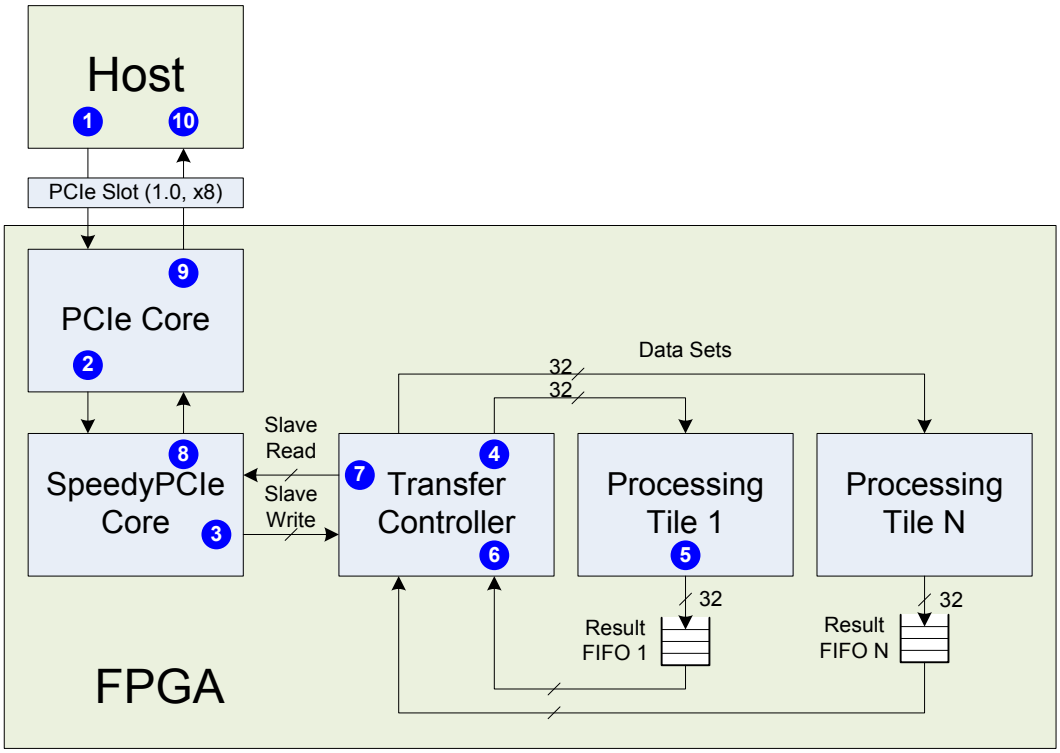

Figure 49 – System Level Topology

1. The host runs an application program. The application program requires acceleration and places a request to the FPGA. This request is made from a Slave Write request

[24] Result FIFOs are shown outside of Processing Tiles for easier data flow presentation

carrying Control information and Data which requires processing. An example of Control information is size of the request and a Start flag.

2. The Write request makes it to the PCIe Core (Xilinx IP) instantiated on the FPGA. The PCIe Core takes care of the PCIe protocol low level related tasks.
3. Speedy PCIe (an open source IP) receives the Write data and forwards it to the Transfer Controller. This block handles a number of tasks such as DMA required to successively pass data to and from the FPGA.
4. The Transfer Controller is a block designed for this project. It parses Control information, splits Data to be processed and delivers split Data to Processing Tiles. Each Processing Tile (PT) performs the same processing task; therefore they provide a speedup equivalent to the number of tiles, in this case N. However Tiles do not have to be used for parallel processing; they could run different sub-tasks in a sequential fashion. The focus, in this thesis, in on parallel processing at the PT level.
5. Processing Tiles, developed for this project, are multi-core modules based on Electron processor. Data sent to a Processing Tile is processed and results are output and stored in Result FIFOs. Result FIFOs are temporary storage. They keep results until a Slave Read request is issued by the Host. At the same time Result FIFOs provide buffering elasticity which is required if processing time (nano-code execution path) is different in different tiles (e.g. a result of branch statements).
6. If a Slave Read request was placed by the Host, the Transfer Controller simultaneously reads all Read FIFOs and performs the required bus width conversions.
7. The Transfer Controller forwards the results to Speedy PCIe Core following the Speedy PCIe interface protocol.
8. Speedy PCIe forwards Slave Read data to the PCIe Core.
9. The PCIe Core forwards Slave Read data to the Host.
10. The Slave Read data is stored in the Host's system memory, making it available to the application program.

5.2 Programming

5.2.1 Sequential versus run-to-completion model

There are two fundamental ways to run programs:

- Sequential programming model: a few kernels working on a different program sections in a sequential manner.
- Run-to-completion model: each Kernel is the running program to completion.

Each model possesses different benefits. Electron Processing Tiles could be organized to run ether one however we use the sequential model in this thesis. For more information see Section 4.8.

5.2.2 Black-Scholes Pseudo Code

In order to maximize the use of Processing Tile resources the Black-Scholes equation is divided into 3 separate units. Each unit is hand compiled into a nano-code program. These 3 individual programs are mapped to run on 4 electrons (one complete Processing Tile). Two Electrons run the same program in parallel. In this project we use term hardware Kernel or just Kernel to describe an Electron running nano-code. Therefore there are 4 Kernels, 2 of which execute the same program: K1, K2A, K2B and K3. Kernels K2A and K2B run the same program.

Programs are created so that each Kernel will process a part of the problem. Program output is intermediate data used by the next Kernel. Therefore processing is sequential.

In the following text, pseudo-code for computation of the Black-Scholes formula is given.

For Black-Scholes formula nomenclatures used in the following text refer to Section 2.8.3.

Kernel 1 (K1):

sqrtT = sqrt(t);

*d1 = (ln(S / k) + (r + 0.5 * σ * σ) * t) / (σ * sqrtT);*

*d2 = d1 - σ * sqrtT;*

Kernel 2 (K2A and K2B):

This Kernel takes output *d* (d1 and d2) from K1 and computes Φ(d) - Cumulative Normal Distribution (CND) as:

*t = 1.0 / (1.0 + b0 * fabs(d));*

Φ(d) *= rsqrt2pi * exp(- 0.5 * d * d) * (t * (b1 + t * (b2 + t * (b3 + t * (b4 + t * b5)))));*

if (d > 0)

Φ(d)*= 1.0* - Φ(d)*;*

This kernel is instantiated two times, one for output *d1* from K1 (processed by K2A) and another for output *d2* from K1 (processed by K2B). K2A outputs *CND1* and K2B outputs *CND2*. Also note use of the *if* statement above. Electron nano-code supports the branch instruction. It should be noted that K2 is the most demanding due to its serialized nature of computing CND value that closely resembles an unwrapped *for* loop.

Kernel 3 (K3):

*expRT = exp(-r * t);*

*expDT = exp(-d * t);*

price = Φ(d1)** S * expDT* - Φ(d2)** k * expRT*

Output of this Kernel is the final Black-Scholes result – Call Price.

Sequential Kernel processing of the Black-Scholes equation is depicted in the following figure.

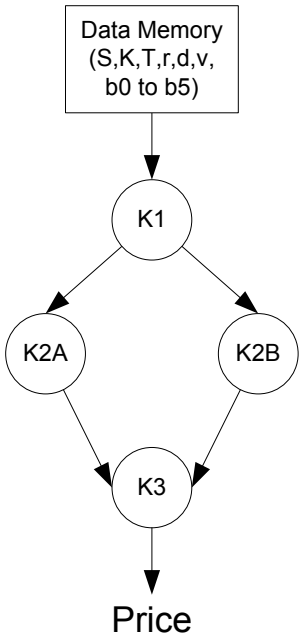

Figure 50 – Black-Scholes mapped to hardware Kernels

5.2.3 Nano-Code VLCW Instruction

Nano-code is a low level program that is loaded into the Electron's Code Memory, see Section 4.5.1. Sections 3.3.6, 4.3.1.5 and 4.3.2.2 respectively explain importance of shorter instruction formats such as smaller instruction memory footprints (comparing to classical VLIW architectures, Section 3.3.6) and wire congestions. The Code Stream unit reads the Code Memory and streams instructions to the Electron's main Controller. The Electron's Controller executes instructions and provides control signals for DynaPath.

The following figure depicts a nano-code VLCW instruction. Each instruction word describes data and control information flow on every clock cycle. We use Kernel K2 as an example (Section 5.2.2). The only difference between Kernels is the number and type of processing elements connected to PXB.

Each K2 instruction controls up to 8 hardware components: 6 floating point arithmetic components, one GPR read/write and one SR read in the same clock cycle. Instructions can have many combinations of instruction fields. They may:

- Instruct the Electron to provide certain inputs to a PE,
- Write a PE output to a GPR,

- Return PE results to the inputs of the same PE or
- Control PXB so that PEs are interconnected in a predefined data flow path.

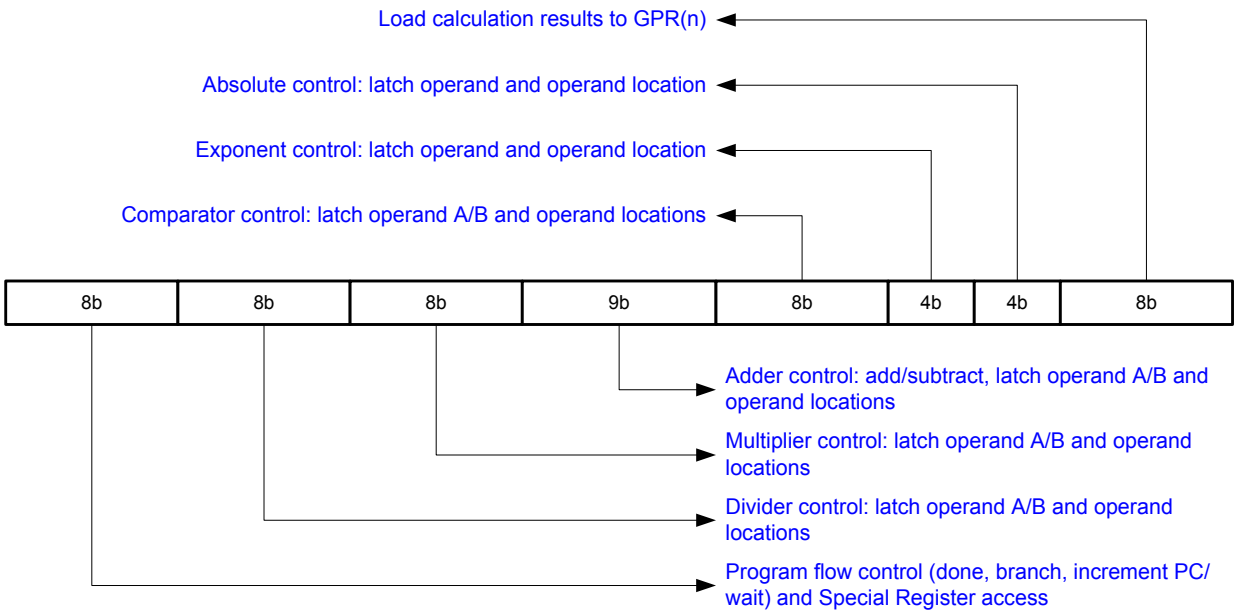

Figure 51 –VLCW example, Instruction Width 57-bits

The Program Flow field controls how the program is executed. This may allow for a Program Counter increment, the execution of a branch or simple wait statement (used to compress program image thereby saving on instruction memory) or the termination of program execution. The branch statement uses the output from a comparator to decide to take or not to take a branch path. K2 executes the following *C* pseudo-code:

if (d > 0)

Φ(d)= *1.0* - Φ(d)*;*

else

Φ(d)= Φ(d)*;*

The equivalent pseudo code:

// bring value *d* to the Compare Processing Element input via PXB and instruct the

// Compare PE to perform graterThanZero comparison. Value 0x0 is stored in SRs:

peCompare (d,0);

// Wait for 2 clock cycles for comparison results:

wait (2);

// change Program Counter based on the compare result:

branch (true - PC=PC1, false PC=done);

The nano-code *wait* instruction, as shown above, is used to clock align PE inputs and outputs. This instruction works same as the *delay* C language command – except the nano-code *wait* delays the execution of the next nano-code instruction following the *wait* instruction for a number of clocks. This may be compared to C language delay represented in time.

Consequently, the *wait* instruction can be used to compress code space. If for example, the program needs to wait for N clocks for PEs to produce results, we can use *wait (N)* – thereby compressing nano-code from N instructions to only one. This in return decreases the amount of Code Memory required.

It is worth noting that the system controller also uses nano-code instructions.

5.2.4 Micro-code versus Nano-code

Microcode is a level of hardware-level instructions used by a higher level machine code program [107]. It allows updates to how the CPU executes instructions without affecting the machine level code. Microcode is used by Complex Instruction Set Computers (CISC). CISC processors contain high speed memory and controller hardware required to translate microcode to the CPU specific hardware logic control signals.

Nano-code is a concept used in this thesis. Nano-code directly controls FPGA hardware; there is no nano-code translation. Nano-code is not translated by the Electron hardware; it gets compiled for the exact hardware.

Figure 52 depicts the differences between the micro and nano-code used in this thesis.

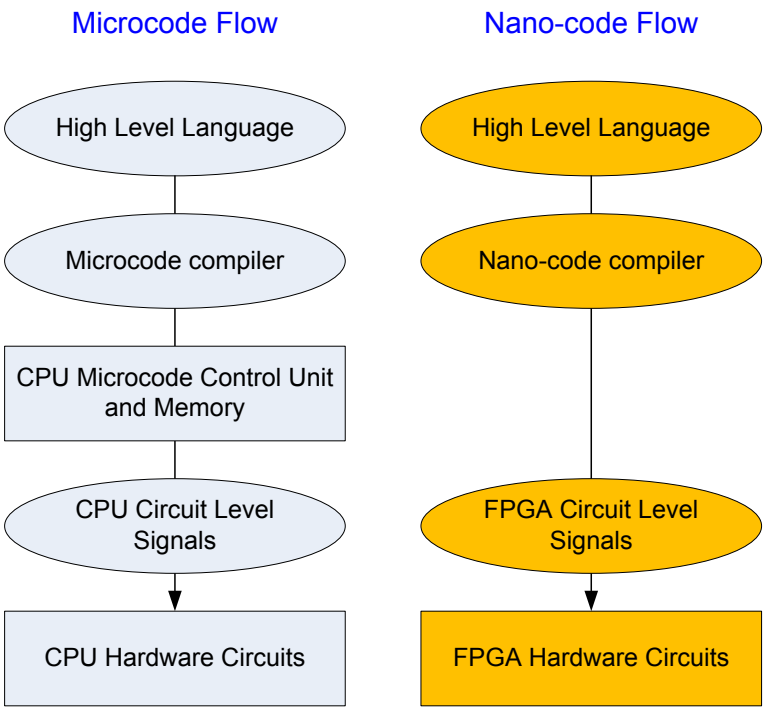

Figure 52 –Microcode versus nano-code

5.2.5 Nano-Code Compilation

In this project the process of compilations to nano-code is manual[25]. The compilation consists of:

- Reducing complex computation problems into smaller computation units. These units are self contained but may use results obtained by other units. Final result of this stage is the pseudo-code, as shown as an example in Section 5.2.2.
- The computation problem reduction may be influenced by the existing hardware. For example each Processing Tile has 4 Electron processors where each processor has access to a certain number and type of processing elements. There could be a number of solutions to the same problem - some could produce lower latency, some higher throughput.
- The pseudo-code computation units are mapped into available hardware for each of Electron processors. This mapping needs to be done in a clock cycle accurate

[25] This fact limits exploration possibilities

fashion. For manual compilation, a spreadsheet is used. This process is very tedious since it involves clock cycle aligning delays from Processing Elements (PE) so outputs from PEs are clock cycle aligned at inputs of another PE. PEs, PXB or block RAM latencies must all be accounted for.

- The last step involves translating human readable tables into nano-code instructions.

5.2.5.1 Kernel Nano-Code Table

In this section we use Kernel 2 as an example. The following clock cycle accurate K2 table, only a section of it shown, is used to create K2 nano-code. Complete nano-code for K2 executes in 136 clocks. Note that rows with multiple consecutive *wait* instructions are hidden (Excel feature) for presentation convenience.

Clock Cycle	Code memory Addr	Operation 1	Operation 2	Operation 3
0	0	read gpr (12)		
1	1			
2	2	mul1_d_d = mul1(GPR, GPR)	abs1_d = abs1(GPR), read gpr (3)	
3	3			
4	4		mul1_0p23_abs1 = mul1(GPR, abs1)	
5	5			
6	wait cnt			
9	wait cnt			
10	6	read special reg (0.5)		
11	7	mul1 port A = special reg (0.5)		
12	8	mul1_m0p5_mul1 = mul1(0.5, mul1)		
13	9		read special reg (1.0)	
14	10		add1_1p0_mul1 = add1(1.0, mul1)	div1 port A = special reg (1.0)
15	11			
16	wait cnt			
17	wait cnt			
18	wait cnt			
19	12		K = div1_1p0_add1 = div1(1.0,add1)	
20	13			
21	wait cnt			
22	14	exp1_mul1 = exp1(mul1)		
23	15			
24	wait cnt			
44	wait cnt			
45	16	read gpr (11)		
46	17			
47	18	mul1_RSQRT2PI_exp1 = mul1(GPR, exp1)		read gpr (10)
48	19			
49	20		*Opera.2 done,* K value from div1 to GPR(15)	mul1 = mul1(div1, GPR)
50	21			
51	wait cnt			
55	wait cnt			
56	22			read gpr (9)
57	23	*Operation 1 done,* mul1 to GPR(13)		
58	24			add1 port A = GPR
59	25			add1=(add1, mul1)
60	26			
61	wait cnt			
62	27			read gpr(15)
63	28			
64	29			mul1 = mul1(GPR, add1)
65	30			

Figure 53 –K2 nano-code

From Figure 53:

Clock Cycle column gives an exact clock cycle when an operation (or operations) takes place. This operation can be fetching of operands (for example in clock cycle 0 read GPR from address 12 - as shown above) or scheduling a PE output and connecting it to a PE input (see column *Operation 2*, clock cycle 14 where an adder uses Special Register containing operand 1.0 and adds it to the output from a multiplier. The Special Register read request was made in clock cycle 13).

Code memory Address is a physical address where a VLCW instruction is stored. Note that from the table, if *wait* instruction is used, this column is not incremented, therefore code-compression is performed.

Operation 1/2/3 columns show all operations taking place in parallel. Number of columns used is determined for clarity and convenience purposes.

The goal of the table is to provide as many operations as possible for the available hardware without breaking any computation/operation dependencies. From it, it also becomes obvious what PEs would benefit from being multi-threaded. PEs used many times should be multi-threaded. The multi-threading option significantly enlarges a Processing Element size (Section 4.6) yet it may reduce processing time.

For example the multiplier is used in clock cycles 2, 4, 12, 47, 49 and 64. The multiplier latency is 10 clocks and a new operation can be executed on every 2nd clock. Therefore the multiplier is generated as a multi-threaded component while the nano-code table is constructed to take advantage of the multiplier's features.

On the other side the divider is used only once in this kernel (clock cycle 19). Therefore, having a divider that offers multi-threading would be wasteful.

5.2.5.2 System Controller Nano-Code Table

It was previously indicated that the System Controller provides all PT top level (DMEM, Kernels and router) scheduling. The System Controller is built as an Electron processor therefore it runs nano-code.

The System Controller:

- Reads DMEM providing equation constants as well as data to be processed to all kernels. DMEM data is distributed to kernels via PT routers (Section 4.4) and written to kernel GPRs.
- Once all data is distributed to kernels, kernels are booted in a sequence required for serial kernel scheduling.

- K1 results are read from K1 GPR and written to K2A and K2B GPRs. K2A and K2B results are read from relevant K2A/B kernel GPRs and written to K3 kernel GPRs. Once K3 is done, the final result is sent out from this PT.
- The System Controller is capable of running a selectable number of loops. In each loop a new set of data is processed and new results created. The number of required loops is determined by the nano-code compiler and it is part of the System Controller's nano-code.

5.2.6 System and Kernel Scheduling

It was previously explained that in this project a sequential programming model is used (Section 4.8.2.2). Kernels 2A and 2B run in parallel (Section 4.8.2.1) – they are part of a sequential model offering speedup. Individual kernels process input data, create intermediate data and feed it to the next kernel in sequence. As soon as a kernel is done processing it takes a new set of data and the processing cycle starts from the beginning.

A scheduling problem arises if two consecutive kernels, kernel N and M, where N proceeds M, have different processing time. If kernel N is faster than M, it cannot start processing a new set of data before M can accept them. Consequently, Kernel N cannot take advantage of its faster speed. Obviously dividing computation problems into appropriate kernels is of crucial importance.

This problem could be remedied by introducing rate adaptation FIFOs between N and M – therefore N could keep processing at its speed. This approach would create no benefits if N and M are part of the same processing algorithm – the slowest kernel determines the speed of all kernels. However freeing up kernel N sooner and repurposing it for a different task, if the processing algorithm could allow this, would improve system processing time.

Figure 54 shows all three Black-Scholes kernels, their interrelation and the equivalent processing time. Color coding is used to indicate time of consecutive kernel runs.

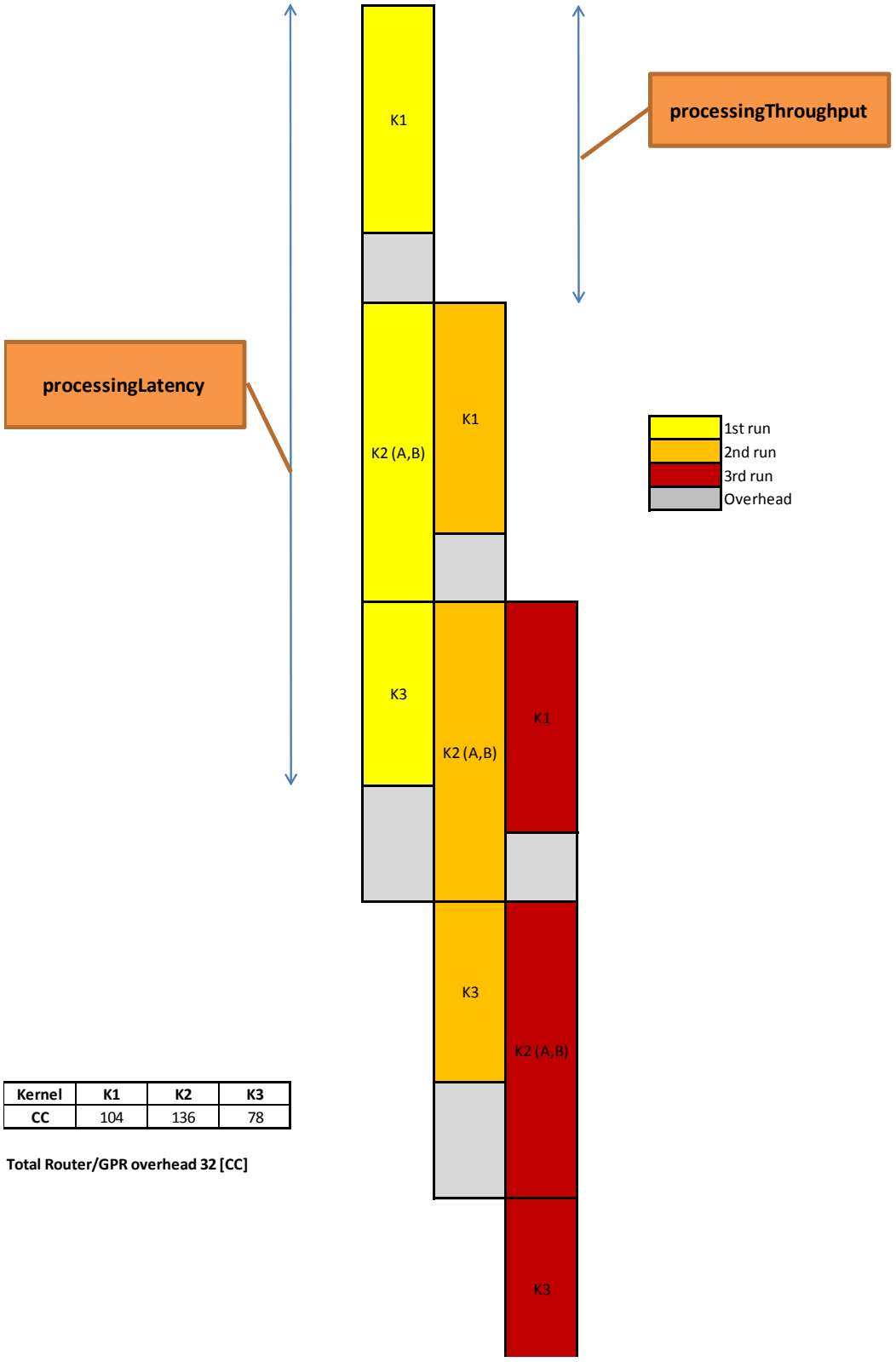

Figure 54 – Processing Tile Kernel Scheduling

5.2.6.1 Processing Throughput

The processing throughput is defined as a number of completed operations in time. An operation is one evaluated Black-Scholes equation:

$$processingThroughput = \frac{frequency}{(longestKernel + overhead)}$$

frequency is FPGA clock rate, 200MHz. *longestKernel* in clock cycles is defined by K2, the *overhead* is time expressed in clock cycles taken to:

- Set all router connections,
- Transfer all processing data,
- Transfer intermediate data from a kernel to next kernel via routers,
- Time required to write and read from GPRs

$$processingThroughput = \frac{200\ MHz}{(136 + 32)}$$

Therefore PT throughput is:

$$processingThroughput = 1.19\ [MOPS]$$

[MOPS] here stands for one Black-Scholes calculation. The above values do not include PC to/from FPGA data transfers delays.

5.2.6.2 Processing latency

Processing latency is the time required for one set of inputs to be evaluated and the time when a result is available at the output. Latency is driven by the Kernel scheduling (and this in return is driven by the longest Kernel processing time) and overhead.

$$processingLatency = ((N - 1) \times longestKernel) + lastKernel + overhead$$

N is the number of Kernels, in this case 3 serial Kernels:

$$processingLatency = ((3 - 1) \times 136) + 78 + 32$$

$$processingLatency = 372\ [CC]$$

Therefore it takes 372 clocks for results of the evaluated Black-Scholes equation to show on PT output port. This number does not include PC to/from FPGA data transfer delays.

5.2.6.3 Summary

Processing latency as well as the equations provided for processing throughput indicates that the kernel with the longest processing time becomes the main performance constraint (Section 4.8.2.2).

K2 takes the longest time due to its serialised nature of iterative multiply and add operations (Section 5.2.2). Adding more computation resources would not improve speed; selecting *add* and *multiply* PEs with the lowest latency available is the key.

When applicable, serializing short dependent kernels and running them in parallel with an independent and long running kernel would be beneficial. If K1 and K3 could run in parallel with K2 the above calculated throughput and latency would be almost 2 times better.

Kernels are interconnected with the 2D-Mesh NoC. NoC router delay, kernel GPR access or DMEM access time add to the processing overhead. In this project routers are designed for minimum pipeline delay.

If applicable, calculations with the longest delay, such as exponential or divide operations, should be initiated as soon as possible. Due to the length of these operations, dividing the computation problem into processing kernels based on the longest operations should be considered.

5.3 System Components – Throughput Overview

The following figure depicts the system throughput capabilities. It was indicated in Section 2.11 that Speedy PCIe core limits the system performance due to its architectural choices. Section 6.4 lists available steps to improve the system speed.

ML605 board supports the PCIe 1.0 x8.

PCIe Fabric is a device placed on a PC motherboard. In this project PCIe Gen 1 with 8 lanes is used. Given 2.5GT/s (Giga Transfers per Second) throughput per lane, it provides 16

GT/s duplex throughput for all 8 lanes. There is 20% overhead due to 8/10 coding that is already removed from this number.

PCIe Core is a Xilinx IP instantiated on the FPGA. With 8 active lanes it produces 1600 MB/s (Mega Bytes per Second). This number needs to be reduced by PCIe transport layer protocol overhead. This overhead depends on the transfer size.

The Speedy PCIe is an open core design that allows interfacing with Xilinx PCIe core. The Speedy PCIe is a flexible solution offered for free to the research community. In order to provide enough bandwidth to application logic and to support the future PCIe versions without changing the Speedy PCIe core, the interface exceeds PCIe Gen 1 bandwidth. This is accomplished by widening the interface to application logic (Processing Tiles) to 64 bytes. The interface is running at 200MHz and it is capable of bursting data in 12800 MB/s duplex rates. This exceeds the available throughput from/to PCIe core therefore sustained Speedy PCI throughput is what PCIe Gen 1 x8 can produce and that is 1600 MB/s.

Due to the FPGA size restriction the FPGA application logic is made from 6 Processing Tiles (PT). Input from Speedy PCIe to each PT is 4 bytes wide and each PT receives input data in parallel. Therefore 24 bytes out of the 64 bytes provided by Speedy PCIe are actually used and 40 bytes are not used. The Speedy PCIe core could interface to the maximum of 16 PTs. Therefore SpeedyPCIe offers (1600MB/s)/16 interfaces = 100MB/s per interface.

It was explained in Section 5.2.6.1 that a PT in our experimental setup takes 168 clocks to evaluate one Black-Scholes equation. Each PT needs 6x 4B words to produce one result. Therefore the input throughput requirement per PT is 28.6MB/s. PTs produce one 32-bit result every 168 clocks. Therefore the output throughput per PT is 4.8MB/s.

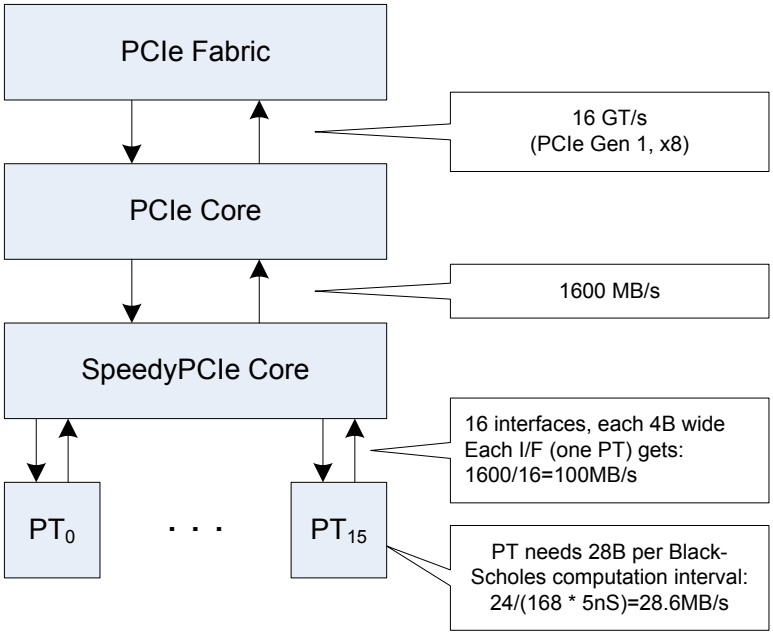

Figure 55 – System Throughput Overview

At this point we conclude that for our experimental setup Processing Tiles are not bandwidth limited by PCIe or Speedy PCIe Core throughputs. One PT gets 100MB/s throughput (minus transaction layer overhead) while PTs need 28.6 MB/s.

However it should be noted the above conclusion is correct for the experimental setup. Section 6.4 explains how and why the experimental setup is limited by ML605 development board (clock rate limitation imposed by SpeedyPCIe). Based on numbers provided in Section 6.4, if the SpeedyPCIe limitation is removed one PT needs close to 96MB/s.

In Section 4.8.2.2 we gave an equation that could be used to calculate number of Kernels for a given workload and input data throughput:

$$N = \frac{Twl}{Td}$$

Let us perform a basic analysis of what would happen if the Black-Scholes equation is divided into a large number of serialized Kernels. In this case PTs will need increased input bandwidth.

Section 5.2.6 shows that total experimental Kernel time is 104 + 136 + 78 = 318 clocks (Twl =318 x 5nS). Let us assume that this 318 clock time does not change if the number of Kernels is changed and that time per Kernel is equal to:

$$timePerKernel = 318/Kernels$$

Therefore the required input data bandwidth is:

$$BWrequired = \frac{6\ x\ 4B}{timePerKernel} = 24B\ \frac{Kernels}{318\ x\ 5nS}$$

From Figure 55 BW offered by SpeedyPCIe is 100MB/s.

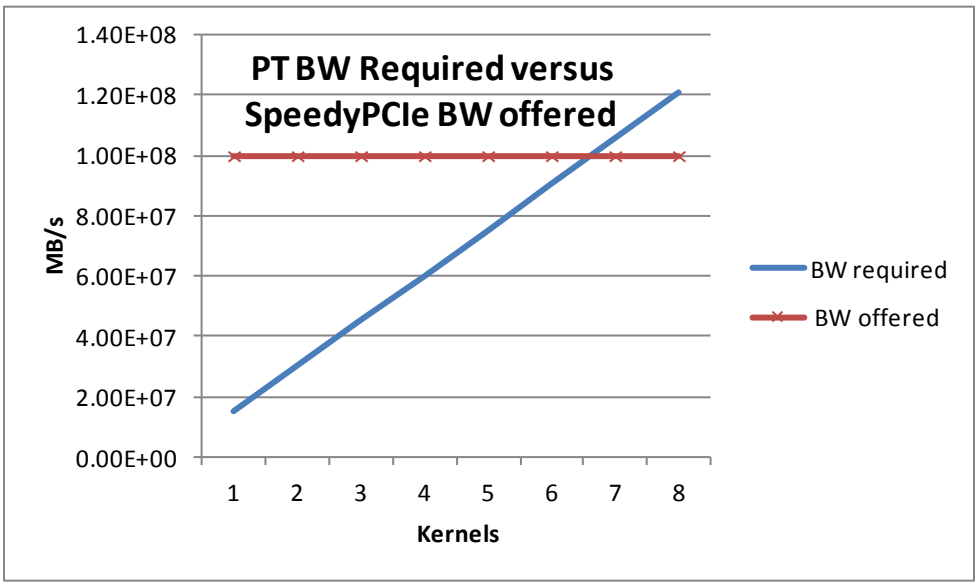

Figure 56 – Kernel BW required versus Speedy PCIe BW Offered

Figure 56 illustrates how PT input bandwidth requirements increase with the number of serialized Kernels. Even for the experimental setup if the number of Kernels is 7 or more, PT input bandwidth requirements (BW required) exceed what SpeedyPCIe can offer (BW offered).

5.4 Backend

5.4.1 FPGA Resources

Processing Tiles run at 200MHz clock rate. This clock rate was selected based on Speedy PCIe core. The following figure gives an overview of resources taken by all FPGA PCIE cores (PCIe Core, Speedy PCI core) as well as by 6 Processing Tiles.

Resources Show Graph

RTL Estimation | Synthesis Estimation | Netlist Estimation | **Implemented Utilization**

Part: xc6vlx240tff1156-1

Resource	Utilization	Available	Utilization
Register	126437	301440	41%
LUT	127911	150720	84%
Slice	35944	37680	95%
IO	45	600	7%
Bonded IPAD	6	62	9%
Bonded OPAD	4	40	10%
RAMB36E1	117	832	14%
FIFO36E1	18	832	2%
RAMB18E1	6	832	1%
FIFO18E1	1	832	1%
BSCAN	1	4	25%
GTXE1	8	20	40%
IBUFDS_GTXE1	1	12	8%
MMCM_ADV	2	12	16%
PCIE_2_0	1	2	50%
BUFG	10	32	31%

Figure 57 – FPGA Resource Utilisation

5.4.2 Application Logic – 6 Processing Tiles

Resource utilization for 6 Processing Tiles:

FLOP_LATCH 113651

LUT 119180

MUXFX 5365

CARRY 97506

For illustration purposes, 6 Processing Tiles take 113651 flops or 1/2 of all used flops, and 119180 LUTs (29795 Slices), or 93% of all used LUTs

5.4.3 Processing Tile

Resource utilization for 1 Processing Tile:

FLOP_LATCH 18194

LUT 19654

MUXFX 889

CARRY 16083

Table 7 depicts resources taken by PEs:

Table 7 - Processing Elements Resource Usage

Processing Element / Parameter	Logarithm	Compare	Divider	Adder	SQRT	Multiplier
Flops	1623	8	404	230	316	679
LUT	1541	61	501	521	336	623
Carry	2036	42	281	319	223	1211
Others	238 (DMEM)	-	16 (DMEM)	28 (MUXFX)	14 (DMEM)	43 (DMEM)

If all PE resources are added together and multiplied by 4 (every PT uses 4 PXBs) we see that approximately 72% of PT resources are from PEs.

In Section 4.6 it was explained that FPGA resources taken by algorithmic primitives directly correlate with the latency and throughput. In this project all primitives except the multiplier were generated for low resource utilization (without adversely affecting the system performance).

The fact that the PEs, despite being generated for low resource utilization, still take almost ¾ of a PT tells us that the System and Electron controllers, their respective crossbars and

PE wrappers are not significant resource contributors. Furthermore if new PEs are to be added, the controllers would take even lower percentage of PT resources for two reasons:

- PEs are the major resource contributors and
- Controllers do not scale in size linearly with new PEs – and that is one of the benefits of this implementation.

5.4.4 Summary

The previous sections indicated that PEs take close to 72% of a PT. In Section 5.2.5.1 a clock cycle accurate table was given showing how PEs are used. On some clock cycles we have 3 operations running in parallel utilizing 3 PEs. However some PEs are used only once during the program run time. For example the logarithm PE is used once only for the duration of 26 clocks over the program's run time of 104 clocks. These numbers would not lead to a good compute density [108].

Obviously a good way to improve the compute density is to share large and exotic PEs so all Kernels within the PT have access to it. In this case the System Controller would schedule use of shared PEs. Shared PEs would be connected to the System Level interconnect just as the Kernels are via routers.

In this section we do not go any deeper into the subject of the compute density due to its huge solution space. Rather we indicate the compute density as one of the constraints to meet for successful deployment of the Electron architecture in the Cloud.

Chapter 6 - Experimental, Achievable and Theoretical Performance

In this chapter we present experimental throughput performance obtained on an FPGA development board and experimental throughput performance obtained on a CPU. FPGA and CPU run respectively hardware and software applications of Black-Scholes computations. FPGA and CPU throughput are compared.

Next we analyse and present opportunities to speed up the FPGA design if a number of limitations, such as ML605 hardware specifics, are removed. We will only be considering improvements with an experimentally or theoretically quantifiable contribution. They lead us to **achievable** FPGA throughput. The FPGA and CPU throughputs are compared one more time. Finally the **maximum**, or theoretical, throughput is presented. This maximum throughput presents an upper bound we would like to approach.

Finally we measure FPGA and CPU power consumptions and compare the two.

Equipment used:

FPGA - Xilinx ML605, Virtex 6 based development board. PCIe Gen 1.0 x8 is used. A set of 6 Processing Tiles was successfully implemented. In total there are 24 Kernels on the FPGA running at 200MHz clock rate.

PC - Gigabyte Z77-HD3 motherboard with 8-GB system RAM, processor i5-3570 running at 3.4GHz - a contemporary PC per today's measures.

Experimental Measurements - Methodology:

A performance measure used is Million Operation per Second (MOPS) achieved by FPGA or PC where the operation is one execution of Black-Scholes equation.

In the FPGA application, raw data required by the Black-Scholes formula needs to be transferred from the PC to FPGA. This is followed by the data processing on FPGA and finally results are returned to the PC. Raw data and result transfers between FPGA and PC are part of provided throughput numbers.

6.1 System Information Flow

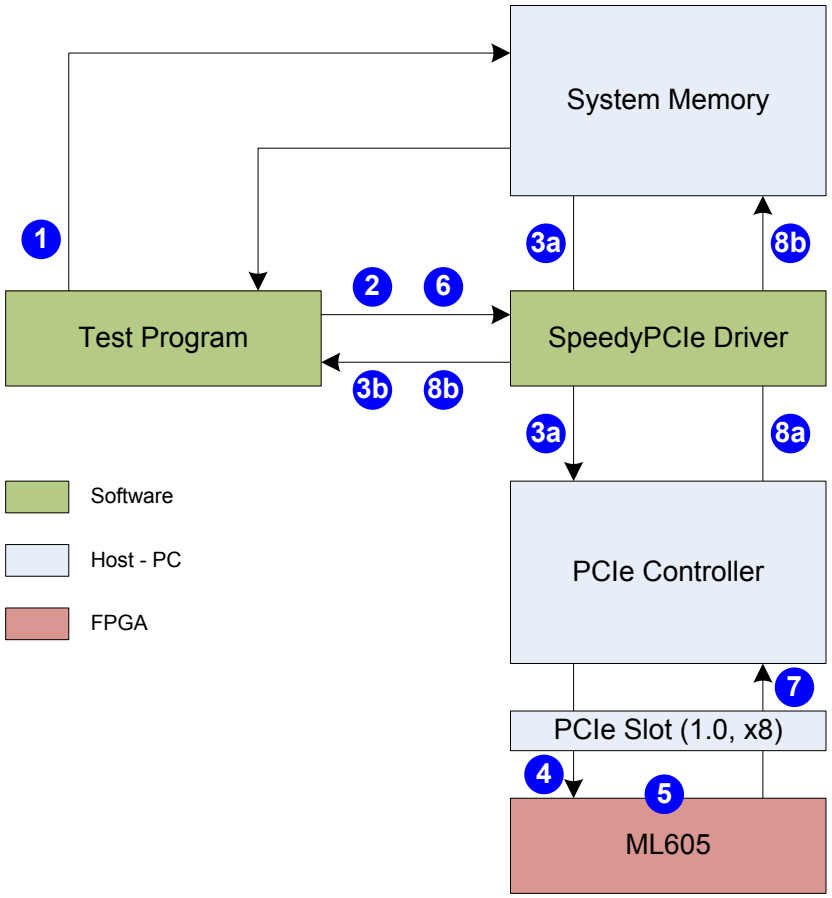

Figure 58 – System Level – Flow of Information

1. Test Program starts. This program populates System Memory with data to be processed. The "data" is made of N x data sets, where each data set will require a result (a Black-Scholes equation evaluation) to be returned by the Transfer Controller residing on the ML605 FPGA back to the Test Program.

2. Test Program makes a Write request to Speedy PCIe driver by passing size of data and a pointer to System Memory where the data is located.
3. Speedy PCIe driver:
    a. Initiates a DMA transfer from the System Memory via PCIe controller all the way down to ML605
    b. As soon as all data is written to ML605 the driver passes program control back to the Test Program by sending the Write request (step 2) status
4. The data makes it to ML605 PCIe core as well as to Speedy PCIe core. The Transfer Controller receives data from Speedy PCIe core, forwards data to all Processing Tiles and initiates processing.
5. Processing Tiles start processing the received data.
6. As soon as Speedy PCIe has returned the Write status (step 3c) the Test Program issues a Read request for the exact number of results as the previously provided data requires. Note that this Read request could not be made until the Write status was provided. Therefore the Write request is a blocking operation. If the processing data was of sufficient size the Read request would arrive while the data sets are still being processed. Since the PCIe interconnect is much faster than the Processing Tiles (Section 5.3), having separate Tx and Rx DMA controllers would not change throughput numbers on large data sets but could on small data sets.
7. The Transfer Controller (FPGA hardware block, see Figure 49) sends results back as a part of DMA response to the Read request. The Transfer Controller is sending results back until all data sets are processed.
8. Next:
    a. The Transfer Controller provided results are forwarded to Speedy PCIe driver by the PCIe controller as part of DMA.
    b. After all results are transferred using the System Memory, the Speedy PCIe driver sends a status to the Test Program. Therefore the Read request is a blocking operation due to how Speedy PCIe driver operates. After a read operation is posted – no new writes can be made until all data has been received.

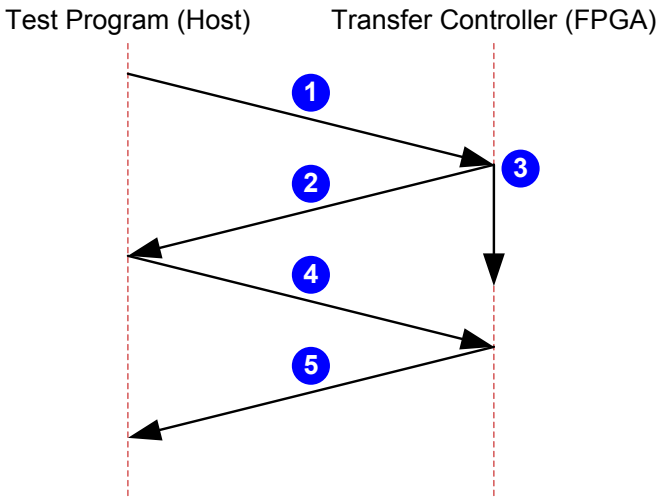

Figure 59 – Hardware-Software Interaction – Case (a)

Case (a), Figure 59: Test Program makes a request to FPGA to process one data set. Transfer Controller works in Low Latency or Low Interrupt Mode.

1. Test Program initiates a Write with one data set (6x 32-bit numbers). Processing Tiles running Black-Scholes nano-code require one data set to calculate one result.
2. ML605 PCIe core acknowledges a Write request and forwards write data to the Processing Tile.
3. The Processing Tile starts processing just received data.
4. Test Program initiates a read request for one result
5. The Processing Tile has completed the processing and is ready to respond to the read request and provide a result via a Transfer Controller. If the Processing Tile did not complete processing, the read request would be delayed and responded to after the Processing Tile is done.

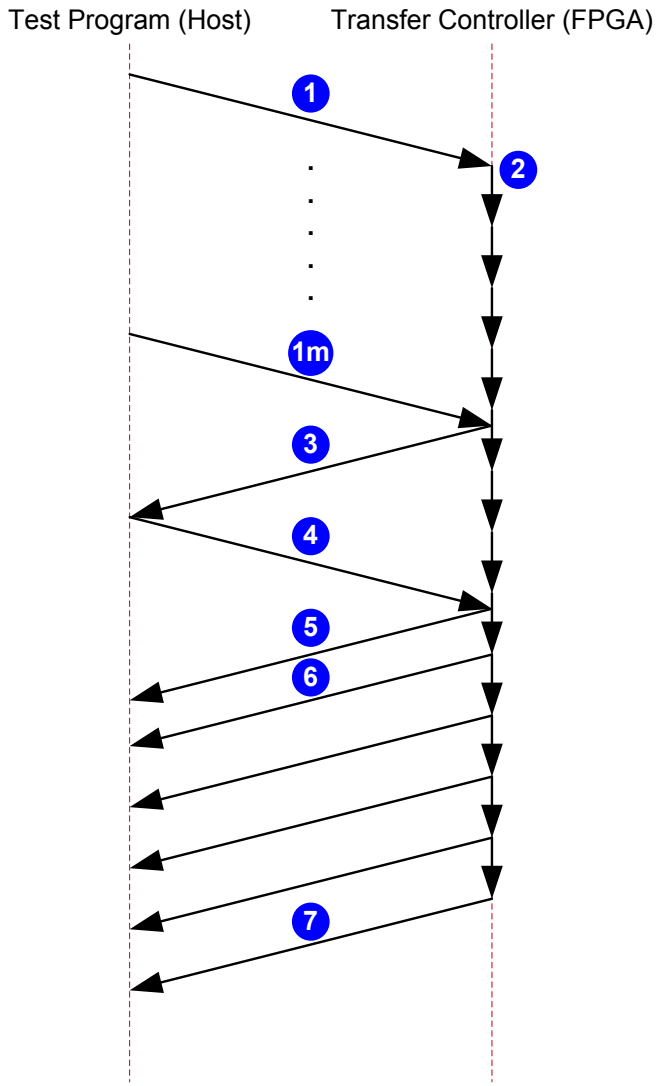

Figure 60 – Hardware-Software Interaction in Low Latency Mode – Case (b)

Case (b), Figure 60: Test Program makes a request to FPGA to process N data sets. Transfer Controller works in the Low Latency Mode.

1. Test Program initiates a Write with N data sets. Each data set will be used by the Processing Tile running Black-Scholes nano-code to create one result. Based on the size of write data, Speedy PCIe driver may choose to send this data in a certain number of transfers. In this example number of transfers is “m”.
2. The Processing Tile starts processing just received data. Each vertical arrow represents creation of one result. Results are stored internally on the FPGA and forwarded to the Test Program after a Read request is placed.

3. ML605 PCIe core acknowledges write request(s) completion
4. After the Write request has been completed the Test Program initiates a read request for N results
5. The Processing Tile responds to Read request by forwarding all results completed to this point.
6. After every new result is generated the Processing Tile sends it to the Host.
7. The previous step continues until all results are forwarded to Test Program

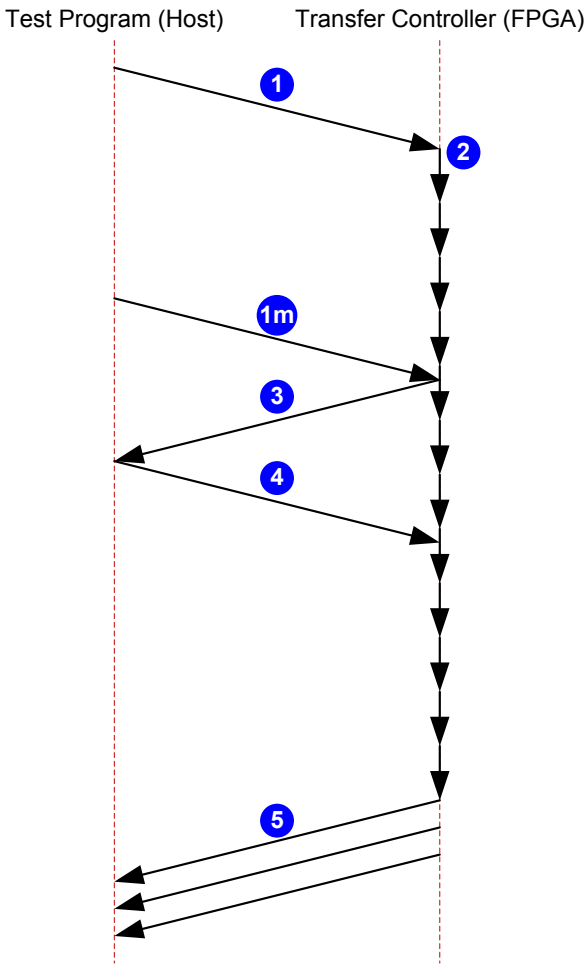

Figure 61 – Hardware-Software Interaction in Low Interrupt Mode – Case (c)

Case (c), Figure 61: Test Program makes a request to FPGA to process N data sets. Transfer Controller works in Low Interrupt Mode.

1. Test Program initiates a Write with N data sets. Each data set will be used by the Processing Tile to create one result. Based on the size of write data, Speedy PCIe driver may choose to send this data in a certain number of transfers. In this example number of transfers is “m”.
2. The Processing Tile starts processing just received data. Each arrow represents the creation of one result. Results are stored internally on the FPGA and forwarded to the Test Program after a Read request is placed.
3. ML605 PCIe core acknowledges write requests,
4. After the Write request has been completed the Test Program initiates a read request for N results,
5. The Processing Tile responds to Read request by forwarding results after all calculations are completed. Number of PCIe transfers is under Speedy PCIe core control. The observed Speedy PCIe transfer size used is a 1k result words which is followed by a turnaround time before the next transfer.

6.2 Read Transfer Modes

The Transfer Controller (TC) was built to support two transfer modes: Low Latency (LL) and Low Interrupt (LI). Modes are selectable during the regular operation by writing to a FPGA TC register.

In LL mode TC reads Results FIFO as soon as there is any data in the FIFO. As shown previously, this approach results in frequent short transfers from ML605 to the Host. Therefore, in this mode results are made available with the lowest latency and in a high interrupt rate.

In LI mode TC reads Results FIFO after all previously requested calculations are performed. In this mode the interrupt level (number of interrupts) is low; however results have somewhat higher latency in arriving at the Host.

6.2.1 Low Latency (LL) Mode

The following figure depicts LL Mode. Reads are requested by the host as soon as raw data is sent to TC (not shown on the diagram). TC sends results (READ_DATA_VALID_H) as soon as results are ready.

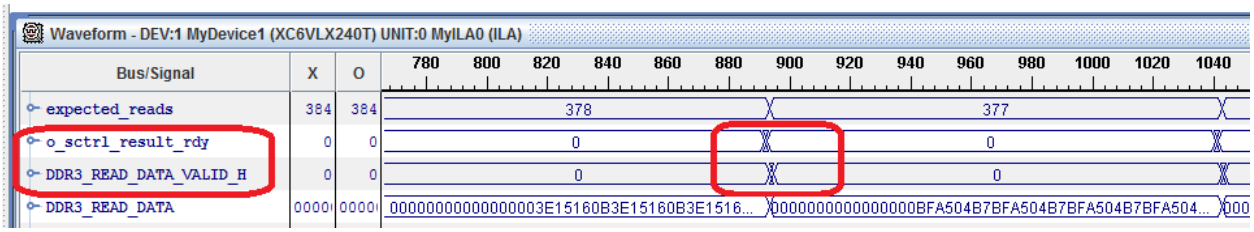

Figure 62 –Low Latency Mode - Waveform

6.2.2 Low Interrupt (LI) Mode

The following figure depicts this case. Reads are requested by the host as soon as raw data is sent to TC (not shown on the diagram) – same as the above LL case. TC sends results (READ_DATA_VALID_H) after all processing results are ready and stored in the Result FIFO.

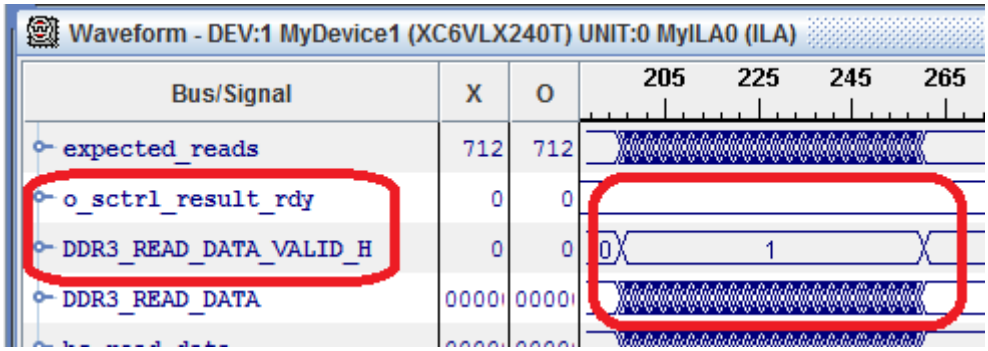

Figure 63 –Low Interrupt Mode - Waveform

6.2.3 Low Latency and Low Interrupt Modes – Comparison

It is expected that LL mode is faster than LI mode. LI mode needs to wait until processing is completed to start sending data to the host. It is portrayed by the following drawing that this expected longer processing due to longer transfer time appears to be within OS noise (jitter).

Figure 64 shows total execution time, including the time when processing data is transferred via DMA to ML605, data is processed and results are sent via DMA transfer back to the host.

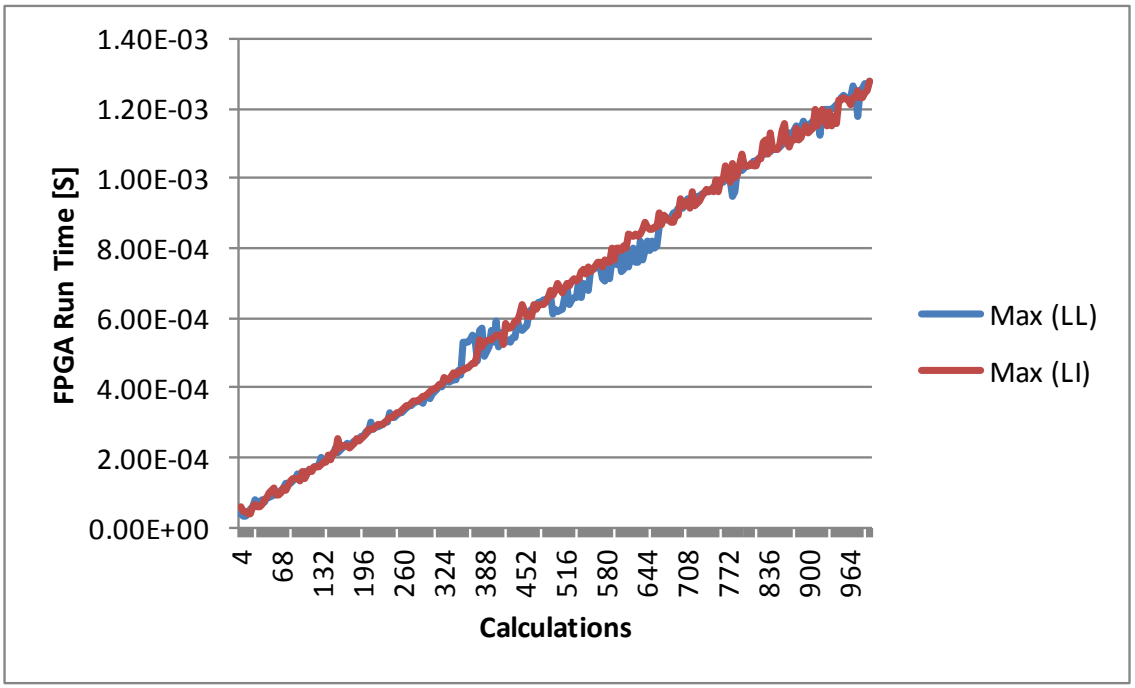

Figure 64 - Low Latency versus Low Interrupt mode

To conclude this section:

- If OS interrupt rate is of concern, LI mode should be used. In many multi-core applications, software developers dedicate one CPU core to interrupt processing. Therefore minimizing interrupts may not be system level constraint of top importance.
- If low processing latency is imperative, as is in financial industry applications, LL mode should be used. How beneficial LL mode is, may depend on the size of raw data (PCIe protocol is more efficient with larger data transfers), PCIe transfer size and other system level constraints. A holistic approach is required.
- LL and LI speed is almost the same (within OS jitter) to process data. The reason for this is that the last result, regardless of the mode, will determine the system

processing speed. Since PCIe as a communication link has higher throughout than Processing Tiles, Processing Tiles themselves will determine the system speed – and that is the last Black-Scholes computation workload. Any calculation before the last, in low latency mode is delivered faster than it would be in the low interrupt mode.

6.3 Black-Scholes Experimental Results

Next we look into software (CPU) and hardware (FPGA) implementations of Black-Scholes equations and compare the two.

6.3.1 CPU Implementation

A C based program was written to emulate the same test bench as it is used for the FPGA implementation. In this test size of the data set is varied from four to 1000 in steps of four. Each data set produces one result – one Black-Scholes evaluation. Ten measurements are performed for each data point. Execution time results are averaged. This approach was taken to minimize variability caused by the Operating System (OS) short term jitter and make Software and FPGA result comparison more meaningful.

A number of the above described tests were performed to gain further insight into how software implementation performance varies. These variations are caused by the long term OS jitter (Section 2.5). In both of the above tests the number of Black-Scholes evaluations is divided by the obtained execution times to arrive at the processing throughput (or just the throughput), expressed in Million Operations per Second (MOPS) of Black-Scholes computation.

A further set of experiments was done running the software Black-Scholes application with the intention of determining if and how the Intel CPU is scheduling calculations. The Intel processor used is i5-3570, a 4 core processor running at 3.4 GHz. If the CPU turbo boost mode is enabled, the CPU clock rate can go as high as 3.8GHz.

The test program is written in Visual Studio 2010 using OpenMP libraries, Operating System is Windows 7, 32-bit. The test program is compiled to run with one, two, three or four threads. At the same time the turbo boost mode is turned On or Off.

The following four snapshots show how i5-3570 is loaded if one, two, three or four threaded Black-Scholes program is run. From it notice that CPU usage increases for approximately 25% for each new thread being turned on.

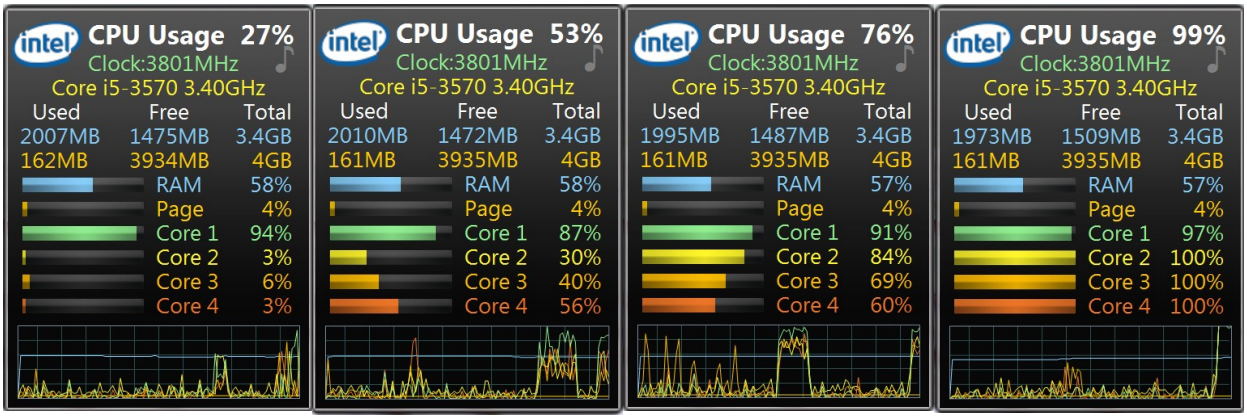

Figure 65 – Intel i5-3570loading[26] for one (first on the left), two, three and four threads enabled

Figure 66 displays the best case (BC) and the worst case (WC) observed software achieved throughput if only one thread (one CPU core) is used (shown as "THs=1", number of threads is one).

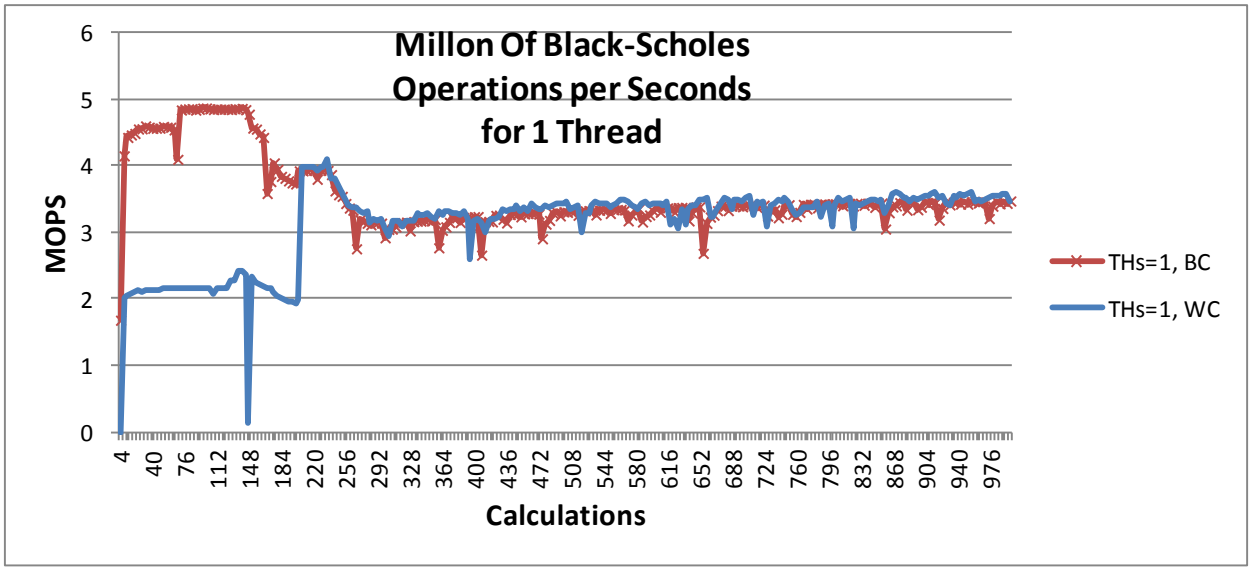

Figure 66 – Best and Worst Case Processing Throughput, turbo boost mode On

[26] Captured by "All CPU Meter" V4.7.3

For the single thread the throughput approaches 3.5 MOPS if the number of performed calculations is above 350. In this region, the OS short or long term jitters effect appear to be negligible.

Figure 67 shows the Black-Scholes throughput if one (THs=1), two, three or four threaded Black-Scholes program is run.

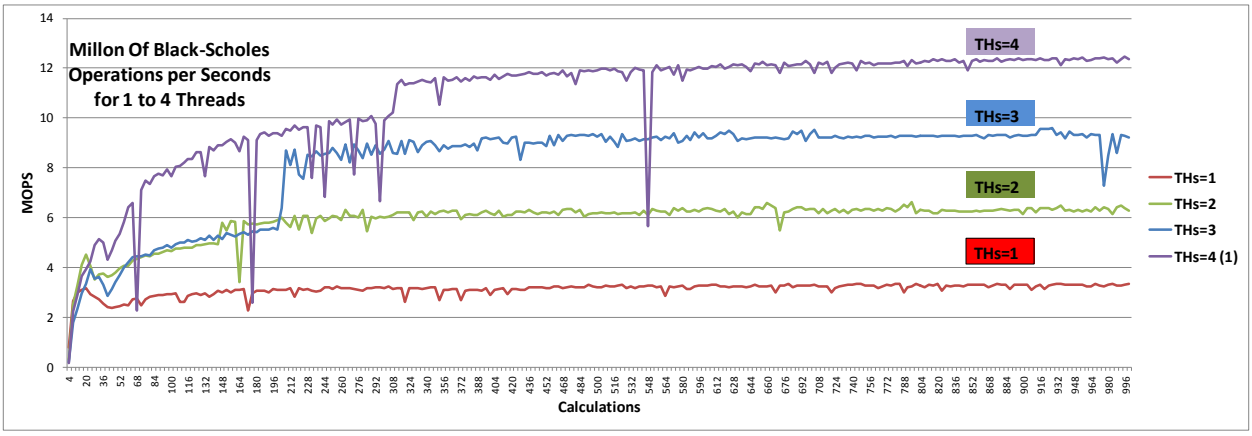

Figure 67 – One, two, three or four Threaded Black-Scholes Processing Throughput, turbo boost is On

Observations:

- One, two, three or four threaded programs create 3.5, 6.3, 9.3 and 12.3 MOPS respectively.
- Three or four threaded programs have somewhat slower turn-on time, notice the step shaped throughput curve. This could be contributed to by the multi-threaded program scheduling and/or the CPU clock rate still being switched between 3.4 and 3.8GHz. The four threaded program has this transition longer than three threaded program.
- The variance of execution time of the three threaded and especially the four threaded programs are larger than the single threaded program. A significant drop in performance is noticed for the four threaded program, shown in Figure 68.
- Performance increase going from a one, two, three to four threaded programs is not an integer multiple of the single thread throughput. Reason for this is that the processor would keep the clock rate high if only single thread is active (turbo-boost

mode is turned on, clock rate is 3.8 GHz) while for three of four threads the processor would be switching between 3.4 and 3.8 GHz to keep the processor die temperature under control.

Figure 68 shows variations in the four threaded program throughput; three runs are shown. The initial step-like turn-on time is caused by the processor thread/core scheduling and CPU clock rater while the performance drops are likely caused by the fact that the i5 processor used is 4-core therefore, any OS scheduling activities would cause erratic processor performance.

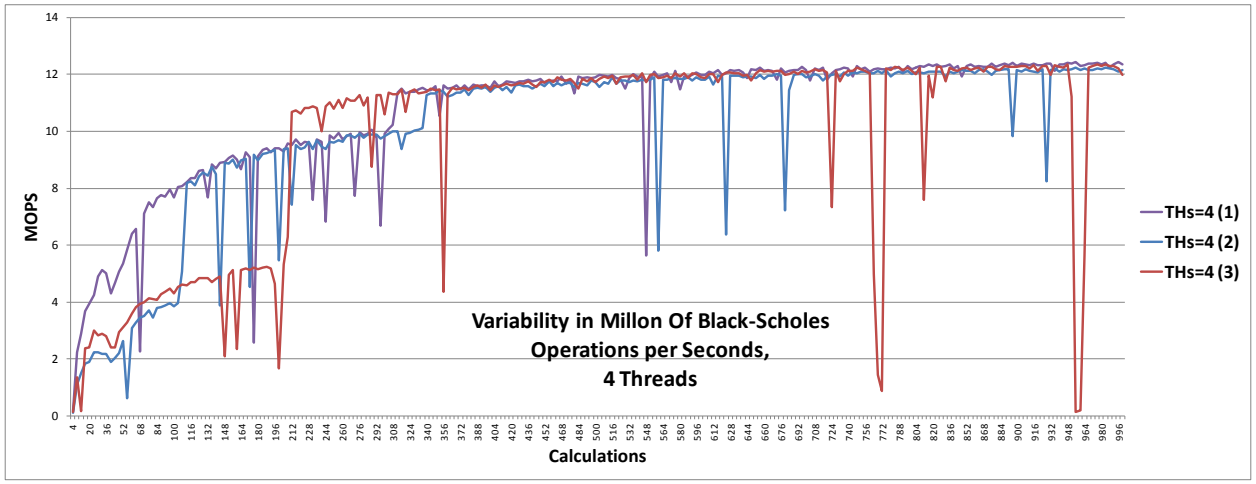

Figure 68 – Four Threaded Black-Scholes Processing Throughput Variation, turbo boost is On

6.3.2 FPGA Implementation

A functionally similar test bench used in software performance testing is created for FPGA. Results for one Processing Tile are provided in Figure 69.

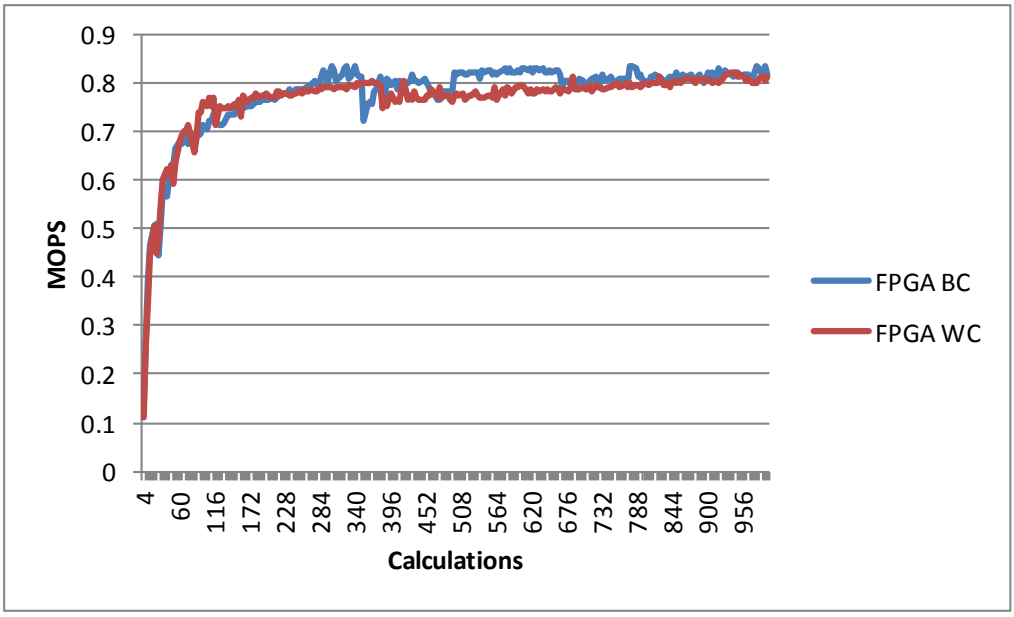

Figure 69 – One Processing Tile Best and Worst Case Experimental Processing Throughput

Contrary to software implementation, processing throughput for the same number of tests constantly displays very low, if any, variability. Worst Case (WC) and Best Case (BC) numbers do not vary by much. Maximum processing throughput is achieved at a somewhat lower number of performed calculations versus software: 300 for FPGA; 350 for software implementation.

Results include all system latencies such as PCIe latency.

6.3.3 CPU and FPGA Implementation – Comparison

In this section we compare processing throughput expressed in MOPS (Black-Scholes computations). Throughput numbers are provided for:

- 1 Processing Tile,
- 6 Processing Tiles and
- Single CPU core best case (BC) and worst case (WC) software implementation.

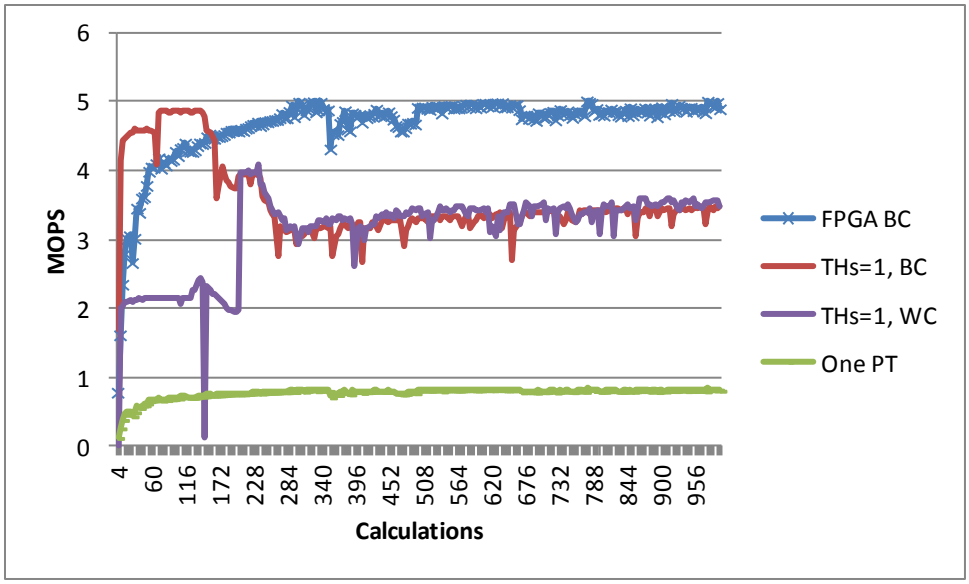

Figure 70 – FPGA (instantiated 6-Processing Tiles), single CPU core software and one PT - Processing Throughput

From the figure:

- The 6-Processing Tile for applications requiring 170+ calculations outperforms the best case corner single core software implementation (FPGA BC).
- Due to CPU processing uncertainties the 6-Processing Tile implementation (FPGA BC) outperforms the worst case software throughput (SW WC).
- The single core CPU implementation is matched by 4-Processing[27] Tile FPGA implementation.

FPGA results include all system latencies such as PCIe latency.

6.4 Achievable System Throughput

The system performance can be significantly improved from what can be obtained on ML605 development board. The following FPGA throughput improvements are available: kernel scheduling and clock rate, adding more Processing Tiles, increased PCIe throughput, lowering PCIe core latencies and software driver consideration.

[27] 4-PT implementation is not shown on the diagram

Throughput improvements obtained are independent – they all can be applied at the same time.

6.4.1 Kernel Scheduling and Clock Rate

The following is a list of opportunities:

- Section 2.11 indicated that Speedy PCIe provides an interface to custom logic that is 512b wide and runs on 200MHz. This clock rate puts an upper bound on the clock rate used by hardware blocks connected to SpeedyPCIe (Electron and Processing Tiles). Based on Electron controller and interconnect design (fully pipelined) as well as Xilinx IPs it is determined that the clock rate could be in the 400MHz – 450MHz range.
- The kernel scheduling shown in Figure 54 could be changed. Figure 54 is just one way that the Black-Scholes equation can be mapped to Processing Tile Kernels. Section 5.2.6.3 lists some of the opportunities. It should be noted that the process of the application partitioning to kernels and nano-code compilation in this thesis are all manual. Therefore experimenting and testing is not easily available.
- Section 2.7 talks about Amdahl's law and what limits the system speedup in a multiprocessor system. It states that the system speedup is limited by a sequential part of an algorithm. Therefore it is of high interest to find and improve a sequential part of Black-Scholes processing. By reducing this time, the system as a whole would benefit.

  Section 5.2 shows that Kernel 2 processes a sequential part of Black-Scholes equation. This part of Black-Scholes takes 136 clocks of which 32 are taken by the division operation (23% of the kernel processing time). In Section 7.3.1 it was suggested to use custom designed processing elements. A similar comment could be made for 4 consecutive multiply and add operations. They take 60 clocks, or 15 clocks per operation. Considering how frequently multiply and add are used, it would be worthwhile to design a more optimised component for commonly used processing.
- Figure 54 shows that the Kernel scheduling has 32 clocks dedicated to system related tasks such as passing processing or intermediate data between kernel GPRs

(23% of the longest kernel processing time). This time could be further reduced. For example, the system controller could distribute processing data ahead of time for more than one run.

Section 5.2.6.1 defined the system processing throughput for the current design. Based on the above opportunities, achievable processing throughput is:

$$achivableProcessingThroughput = \frac{frequency}{(longestKernel + overhead)}$$

$$achivableProcessingThroughput = \frac{400\ MHz}{(136 + 32 - a - b - c))}$$

Where:

a = Clocks removed by improving the division processing elements, here we assume this number would be reduced from 32 to 16 clocks

b = Clocks removed by replacing multiply and add components with a fused multiply-add (such as Xilinx XtremeDSP) component, here we assume this number would be reduced from 60 to 16 clocks (4 multiply-add operations)

c = Clocks removed by lowering overhead due to passing processing or intermediate data, here we assume this number would be reduced from 32 to 16 clocks

Achievable PT throughput is:

$$achivableProcessingThroughput = \frac{400\ MHz}{92\ clocks} = 4.4\ [MOPS]$$

Note that the above *achivableProcessingThroughput* also defines the maximum PT throughput, meaning if other system components (such as PCIe interconnect) were not included, or were much smaller than the *achivableProcessingThroughput,* the *achivableProcessingThroughput* would become the *maximumProcessingThroughput* one PT could offer.

Next we define a Scheduling Speedup Factor (SSF) or improvement in processing throughput as:

$$SSF = \frac{achivableProcessingThroughput}{processingThroughput}$$

$$SSF = \frac{4.4\ [MOPS]}{1.19\ [MOPS]} = 3.7$$

Therefore the system throughput due to the scheduling speedup can be improved by 3.7 times.

6.4.2 Adding Processing Tiles

In this section we determine how to increase the number of processing tiles. There are two direct ways:

- Share selected processing elements by all Kernels within a Processing Tile
- Use a larger FPGA

As indicated in Table 7 of Section 5.4.3, the Logarithm Processing Element with 1541 LUT is the largest of all Processing Elements. There are 4 instances of this hardware block per Processing Tile. Complete PT has 19654 LUTs. If all Kernels shared one Logarithm, PT size would be reduced for 3x or 1541 LUT, therefore a PT with a shared Logarithm would have 15031 LUT.

ML605 development board is built with Virtex 6 XC6VLX240T FPGA device. This particular V6 device is an average size device from the V6 family. Table 8 gives the number of PTs if the Logarithm hardware block is shared per PT (PT LUTs), or not (PT Shared PE LUTs), versus Virtex 6 FPGA device (marked as “base line”):

Table 8 – Adding Processing Tiles

Virtex 6	XC6VLX240T	XC6VLX365T	XC6VLX550T	XC6VLX760
Available Slices	37680	56880	85920	118560
Available LUTs	150720	227520	343680	474240
PT LUTs	19654	19654	19654	19654
PT Shared PE LUTs	15031	15031	15031	15031
Max # of PTs	7 (base line)	11	17	24
Max # of PTs (shared PE)	10	15	22	31
Increase in # of PT	1.00	1.57	2.43	3.43
Increase in # of PT (shared PE)	1.43	2.14	3.14	4.43
Processing Tile Speedup Factor	$PTSF_1$	$PTSF_2$	$PTSF_3$	$PTSF_4$

From Table 8 we see that the maximum increase in number of PTs is 4.43 times for a case when the Logarithm hardware block is shared per PT and if we go from XC6VLX240T to XC6VLX760 device. We call this increase in the number of processing tiles Processing Tile Speedup Factor or PTSF.

6.4.3 PCIe Throughput Increase

ML605 uses PCIe Gen 1.0 x8 as Host system interconnect. Adding more lanes would speed up the most significant system delay component – Host to the coprocessor and back to the Host latency. This latency affects the throughput since it is part of all calculations provided in Section 6.3.

A set of experiments was performed to evaluate how PCIe throughput affects the FPGA Black-Scholes evaluation speed. We define the following two terms:

- Interconnect Speedup Factor (ISF) as a change of the application run time due to the PCIe throughput increase and
- PCIe Throughput Increase Factor (PCIeTIF)

Figure 71 presents experimentally obtained data. It depicts the throughput increase when PCIe Gen 1.0 x4 is changed to PCIe Gen 1.0 x8. Throughput results presented are for

doubled PCIe throughput (PCIeTIF=2, we name it as $PCIeTIF_1$). The Black-Scholes calculation speed (ISF) is increased between 25% (1.25 times) and 100% (2 times). We name this ISF as ISF_1.

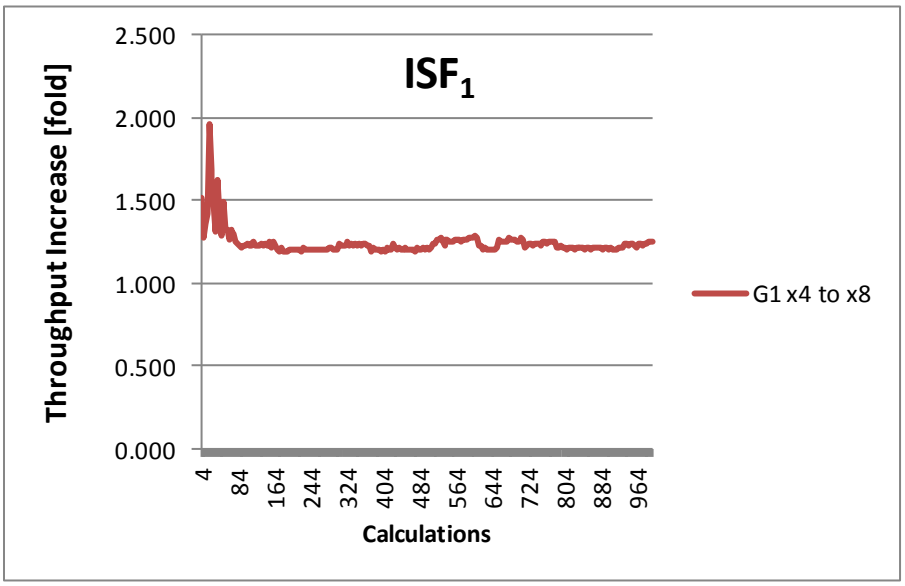

Figure 71 – Black-Scholes FPGA Throughput Increase: PCIe Gen 1.0 from x4 to x8

Armed with the above defined terminology and ISF_1 numerical values from Figure 71 we proceed to estimate ISF if we upgrade the ML605 development board PCIe link from Gen 1.0 x8 to Gen 2.0 x8 (maximum available on Virtex 6 device, equivalent to PCIeTIF=2.5) or Gen 3.0 x8 (maximum available on Virtex 7 device, equivalent to PCIeTIF=4).

The Black-Scholes equation evaluation would increase in speed by:

$ISF_{2.5}$, Gen 1.0 x8 to Gen 2.0 x8:

$$ISF_{2.5} / PCIeTIF_{2.5} = ISF_1 / PCIeTIF_1 \text{ or}$$

$$ISF_{2.5} = ISF_1 \times (PCIeTIF_{2.5} / PCIeTIF_1)$$

Gen 1.0 x8 to Gen 3.0 x8:

$$ISF_4 / PCIeTIF_4 = ISF_1 / PCIeTIF_1 \text{ or}$$

$$ISF_4 = ISF_1 \times (PCIeTIF_4 / PCIeTIF_1)$$

Figure 72 shows the above calculated ISF charts.

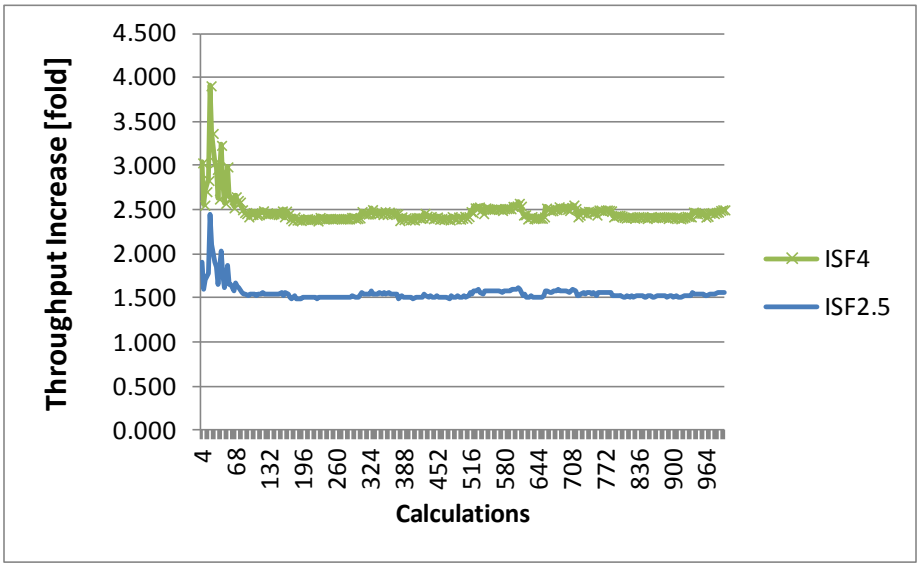

Figure 72 – Black-Scholes FPGA Throughput Increase: PCIe Gen 1.0 x8 to Gen 2 ($ISF_{2.5}$) and 3 (ISF_4) x8

Calculated average numbers are:

$ISF_{2.5} = 1.55$

$ISF_4 = 2.48$

In Section 6.5 we use $ISF_{2.5}$ only.

6.4.4 Xilinx PCIe and SpeedyPCIe Cores

Both of PCIe cores contribute significantly to the system processing latency. Processing data as well as result data need to be received and sent via DMA transfers to the Processing Tiles effectively passing through two PCIe cores twice.

SpeedyPCIe core is free intellectual property created for the research community. Its latency was not necessarily a primary goal.

In this thesis we did not factor in any calculation throughput improvements related to the PCIe cores. Section 6.6 sheds some light on what Electron could do if PCIe latencies are minimized.

6.4.5 Software Driver Considerations

SpeedyPCIe driver is free intellectual property created for the research community. Its speed and latency was not necessarily a primary goal. Some known driver issues are documented in Section 6.1.

In this thesis we did not factor in any driver performance disadvantages.

6.5 CPU and Achievable FPGA Throughput - Comparison

In this section we present achievable FPGA throughput based on the analysis presented in Sections 6.4.1, 6.4.2 and 6.4.3 where it was shown that the processing throughput can be increased by combined SSF, PTSF and ISF ($ISF_{2.5}$ only) speedup factors. For a multi-threaded CPU performance refer to Section 6.3.1.

Figure 73 shows the achievable FPGA throughput for various FPGA families. The ML605 development board uses XC6VLX240T. Only Xilinx Virtex 6 family is considered. The four threaded CPU performs the worst; XC6VLX240T FPGA is just above the CPU while XC6VLX760 has the highest MOPS.

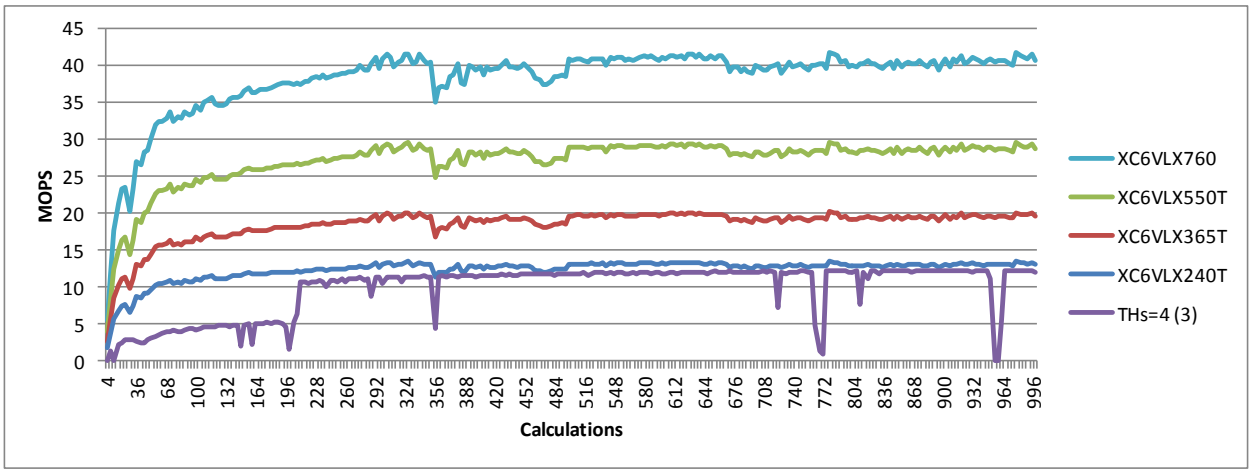

Figure 73 – Achievable FPGA and i5-3570 CPU Processing Throughput

6.6 FPGA Theoretical Throughput

In this section we provide throughput comparison for one Processing Tile for the following cases:

- Achievable – this is the experimentally observed (FPGA) PT throughput with ISF (Section 6.4.3) and SSF speedups applied (Section 6.4.1),
- THs=1, BC – the single CPU thread/core best case (Figure 66 gives the single thread CPU throughput), an experimentally measured throughput and
- Maximum – (FPGA) PT theoretical maximum (Section 6.4.1). In this case the system throughput is not hindered by the PCIe throughput/latency, two PCIe cores and their latency or software driver specifics.

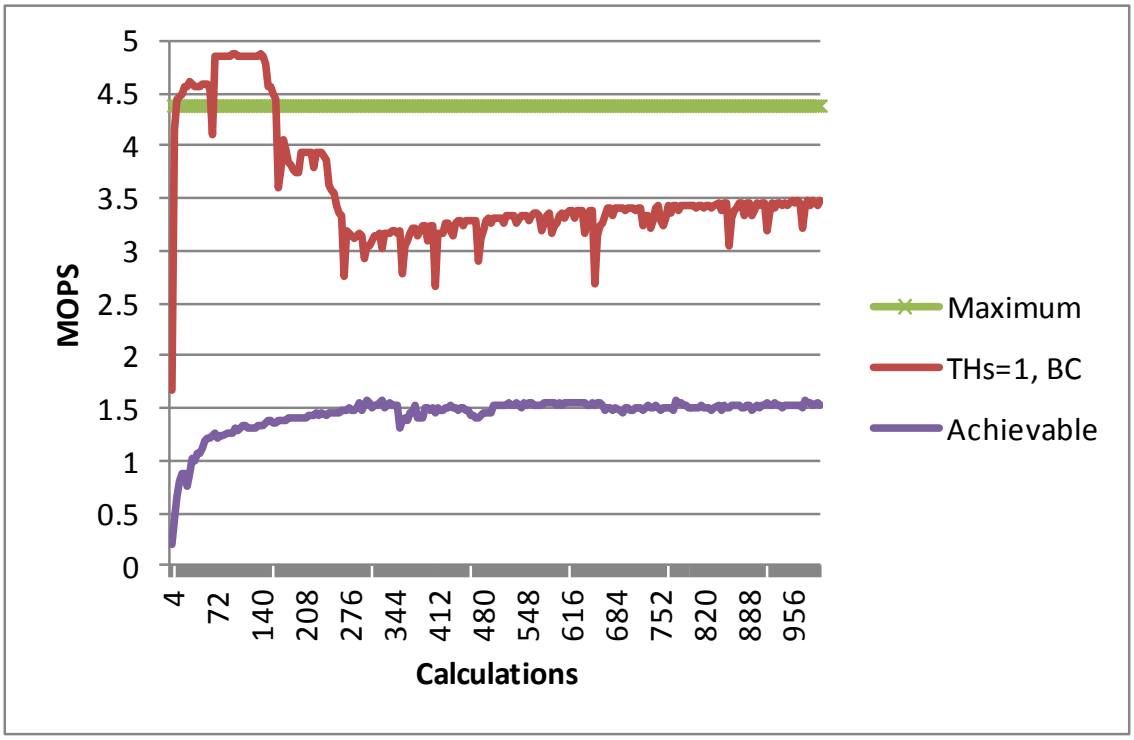

Figure 74 – Single PT (Achievable and Theoretical Maximum) versus Single CPU thread/core Throughput

Figure 74 shows evidence of the potential of Electron Processing Tiles and FPGAs. One PT could offer more MOPS ("Maximum") than a CPU ("SW BC") core.

From the figure we see:

- Cloud applications that require minimum Host – Coprocessor communication are the best candidates for offloading. They could be significantly accelerated and **approach** the "Maximum" performance (examples are applications based on the Monte Carlo analysis, Section 7.3.2).

- Difference between the "Achievable" and "Maximum" numbers is mostly caused by the PCIe cores and the software driver, therefore investing development time in PCIe cores and software driver is worthwhile.

6.7 CPU and FPGA – Power, Performance and Cost

In this section we examine measured power consumed by the CPU and FPGA while processing Black-Scholes equation. Next we calculate "Performance per Watt" and "Performance per Watt per Dollar" for CPU and FPGA followed by the number comparison.

Figure 75 presents Intel CPU power consumption for a single threaded application. The CPU static power is 7W and when executing Black-Scholes calculations power goes up to 26W – an increase of 19W, or 270%.

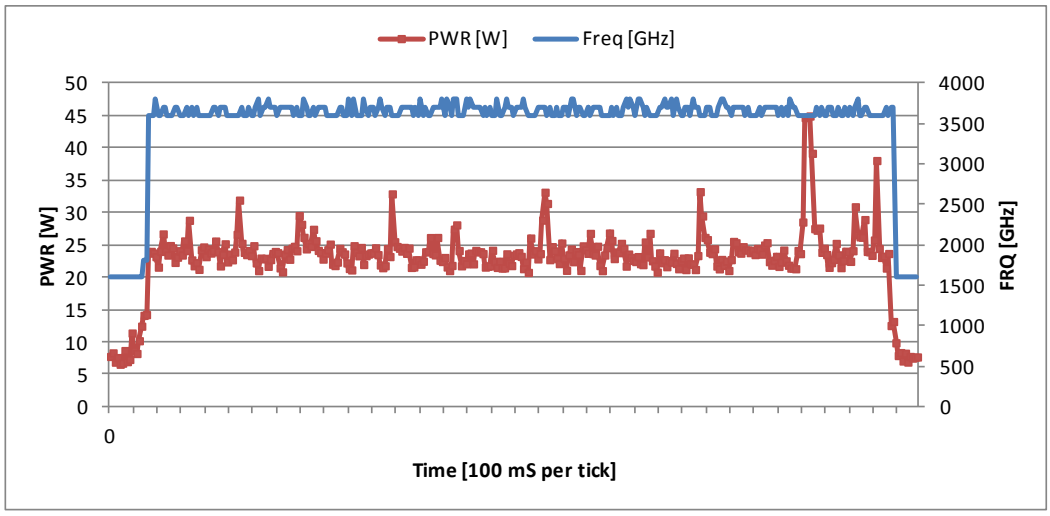

Figure 75 – Intel i5-3570 CPU power and frequency[28] during single threaded Black-Scholes calculations, turbo boost mode is On

Adding a level of multi-threading increases the CPU performance by an amount approximately equal to a single core performance. Refer to Section 6.3.1 for the exact analysis. Figure 76 and Figure 77 show that the CPU power is incremented for a (significant) fraction of the single core power if the level of multi-threading is increased. It

[28] Data captured by Intel Power Gadget tool. For comparison 4 core video conversion takes 43W.

appears that if only a single core is loaded the clock rate for all 4 cores is increased. This would imply that adding multi-threading does not multiply the power consumption by the level of threading, but rather that the power is fractionally increased by the extra processing caused activity rate such as register state toggling, memory accesses and more.

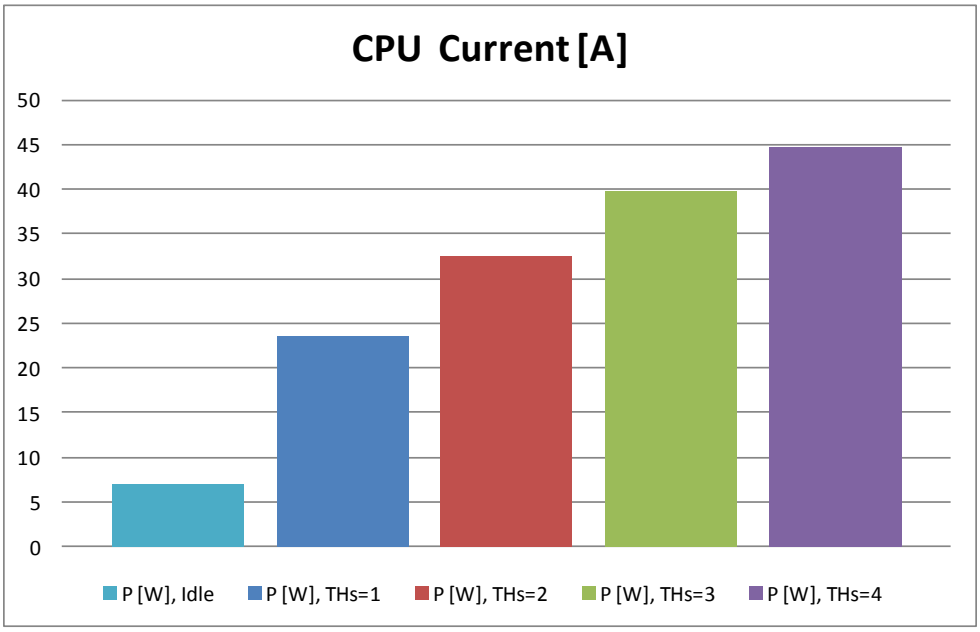

Figure 76 – Intel i5-3570 CPU Power for idle, one, two, three and four threaded Black-Scholes, turbo boost mode is turned On

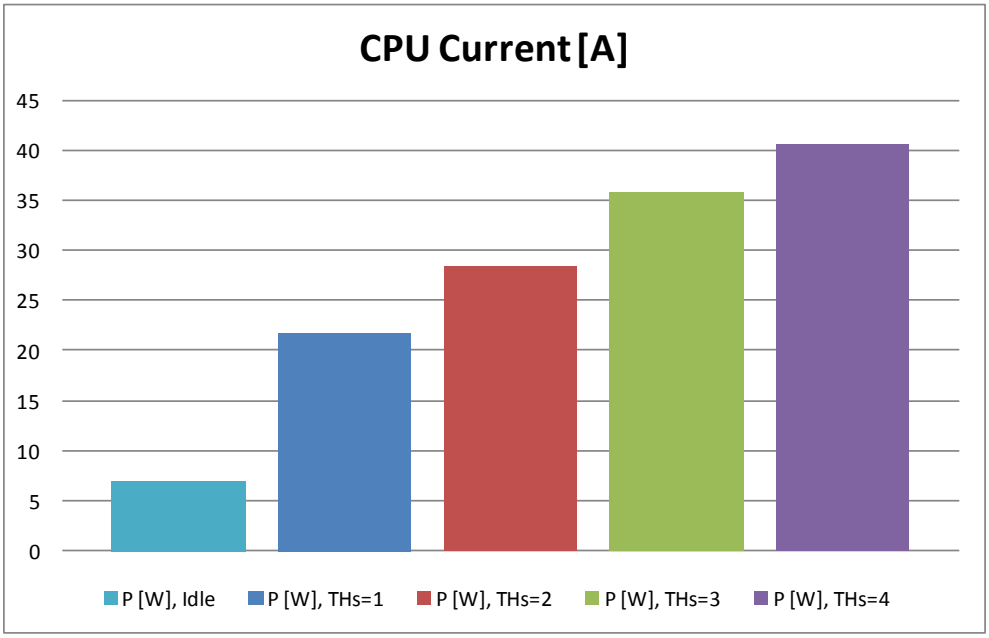

Figure 77 – Intel i5-3570 CPU Power for idle, one, two, three and four threaded Black-Scholes, turbo boost mode is turned Off

Figure 78 shows CPU Performance per Watt (PPW) expressed as number of MOPS of Black-Scholes computations versus consumed power in Watts. Note that PPW is almost identical if the turbo boost mode is turned On or Off. If turned On, the application runs faster but consumes more power.

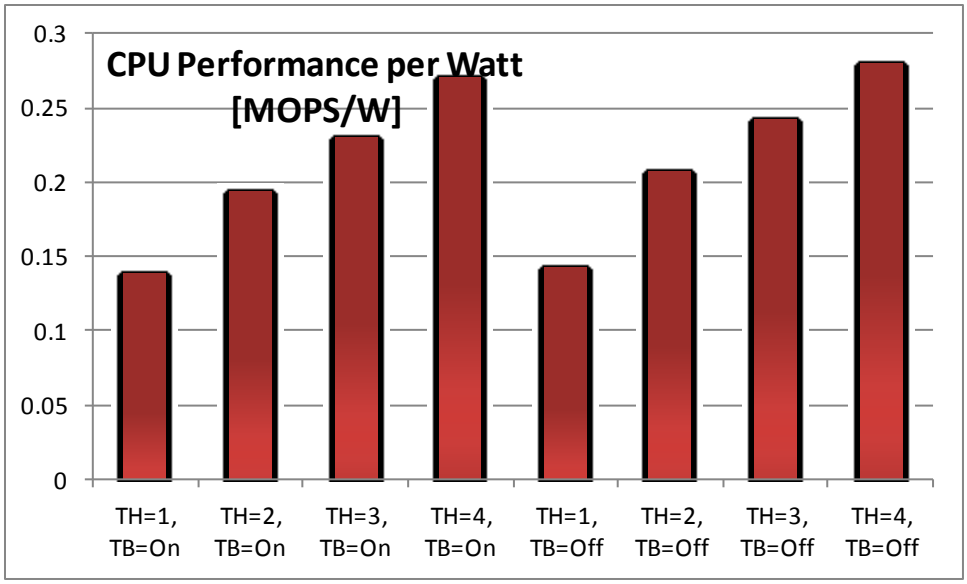

Figure 78 – Intel i5-3570 CPU Performance per Watt for one, two, three and four threaded Black-Scholes, turbo boost mode is turned On/Off

Figure 79 presents measured FPGA power consumption. FPGA static power is 4.35W and when executing Black-Scholes calculations power goes to 4.7W – an increase of 350mW or 7.4%.

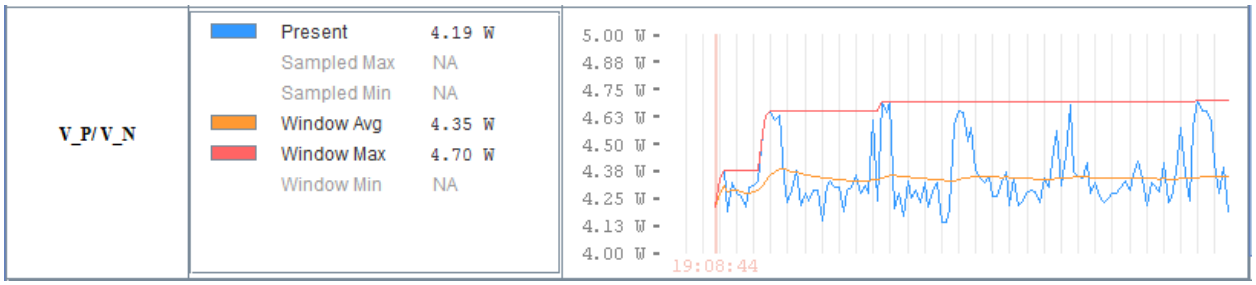

Figure 79 – Xilinx XC6VLX240T power consumed[29] during Black-Scholes calculations

Next we determine Performance per Watt (ppw) for FPGA and CPU and compare the two.

Performance per Watt analysis is done for the section of Figure 73 where FPGA (XC6VLX240T) and CPU MOPS (Million of Operations Per Second, where 1 operation is one Black-Scholes equation computation) curves are constant (668 Black-Scholes calculations).

[29] Captured by Xilinx ChipScope Pro

The CPU Performance per Watt is based on MOPS performance from Figure 73 and power from Figure 76:

$$CPUppw = \frac{12.12\ MOPS}{44.8\ W} = 0.27\ [\frac{MOPS}{W}]$$

The FPGA throughput shown on Figure 73 is based on 400 MHz clock (Section 6.4.1) while the power consumption reported in Figure 79 is for a 200MHz clock Electron experimental implementation. Therefore the power needs to be adjusted to include this increased clock rate.

From Section 2.6 we see that out of three power consumption components only the dynamic part needs to be adjusted. The equation for dynamic power:

$$Pdynamic = \left(\frac{1}{2}\mathrm{C}\ V^2 + \mathrm{Qsc\ V}\right) \times \mathrm{F}\ \times\ \mathrm{A}$$

From the equation only the frequency is changed. It is two times higher; therefore the dynamic power is going to be two times larger. From Figure 79:

$$Pdynamic200MHz = 350\ mW$$

Therefore:

$$Pdynamic400MHz = Pdynamic200MHz\ \times\ 2 = 700\ mW$$

This new dynamic power for 400MHz includes both logic gates and clock trees. The MOPS number is shown on Figure 73. FPGA Performance per Watt:

$$FPGAppw = \frac{13.2\ MOPS}{4.35\ W + 0.7\ W} = 2.61\ [\frac{MOPS}{W}]$$

Next we calculate FPGA to CPU Performance per Watt ratio:

$$FPGAversusCPUppw = \frac{2.61}{0.27} = 9.6x$$

Therefore the XC6VLX240T FPGA solution has 9.6 times better Performance per Watt versus the CPU.

Another way of comparing processors is Performance per Watt per Dollar (ppwp$). In this case:

$$CPUppwp\$ = \frac{CPUppw}{250.00\$^{30}} = 0.00108\ [MOPS/W/\$]$$

$$FPGAppwp\$ = \frac{FPGAppw}{1900.00\$^{31}} = 0.0012\ [MOPS/W/\$]$$

Finally:

$$FPGAversusCPUppwp\$ = \frac{0.0012}{0.00108} = 1.11x$$

Therefore even after the initial investment, an FPGA-based solution still comes ahead of a CPU. This number does not include thermal or power supply costs associated with the CPU solution. If these costs were considered, they would tip the results even more towards the Electron and FPGAs.

6.8 Summary

The Black-Scholes equation is implemented on a CPU and an FPGA. The ML605 development board presented a number of performance limitations.

In this thesis it was identified that at least 20% of applications running in Clouds could be offloaded from CPUs by FPGA technology. These applications are parallel in nature and have high computation-to-communication ratio. The computation-to-communication ratio determines what applications can take advantage of concurrent processing done on FPGAs [6]. FPGA coprocessors need processing data downloaded from the host system to the FPGA and results need to be uploaded from the FPGA to the host system. Download and upload time represent the communication part of the computation-to-communication.

[30] Source: Canadian Computer
[31] Source" Digi-Key

Next analysis of ML605 and used technology (intellectual properties such SpeedyPCIe core) limitations is performed and a set of improvements that would result in throughput increase is determined.

We applied some of the specified improvements to the experimentally obtained data arriving at the achievable throughput. Improvements are selected based on how easily their contribution can be proven and quantified experimentally (PCIe link speed) or theoretically (revisited Kernel scheduling or adding new Processing Tiles).

Remaining improvements are only listed as opportunities for further research (Xilinx PCIe or more likely SpeedyPCIe, next are software drivers). Their implementation is considered to be beyond the scope of this thesis.

To gain an understanding of available processing capacity offered by Electron and Processing Tiles we show what they can do if not limited by the environment they are used in. We call this "the theoretical throughput" or "maximum throughput".

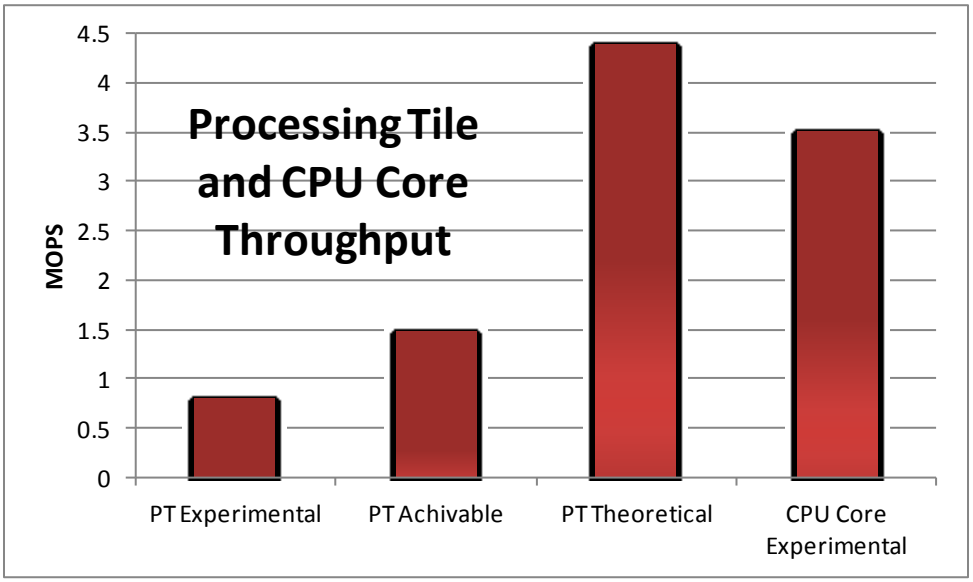

Figure 80 – Single Processing Tile and single CPU core Black-Scholes Computational Throughput

Figure 80 shows the obtained throughput numbers expressed in Million Operations per Second (MOPS) of Black-Scholes computations for a single Processing Tile and a single CPU core. Based on the experimental data (severely limited by ML605 development board), the

FPGA needs to instantiate 4.4 Processing Tiles to be ahead of a single CPU core. FPGA needs to instantiate 2.3 PTs if achievable throughput numbers are used. A single PT not limited by the system (the theoretical throughput used) offers a 26% improvement over a single CPU core. Section 7.3.2 gives an example of an application whose throughput would be significantly higher than "PT Achievable" approaching "PT Theoretical".

The difference between the achievable and theoretical PT throughput is significant (2.9 times). From the system perspective, the host PC cannot ever see this theoretical throughput. However, we use the theoretical PT throughput to signify the performance offered by the suggested technology as well as to see opportunities for further research (Section 7.3.1). Monte-Carlo based applications are very good candidates. These applications do not require a constant stream of processing data from the host.

Next we look into FPGA and CPU power consumption.

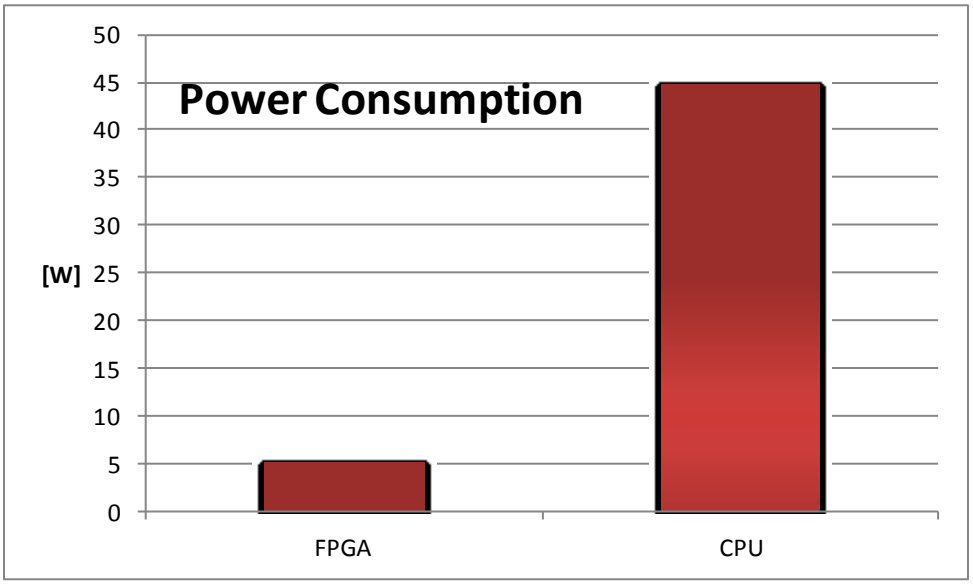

Figure 81 – FPGA and CPU Power Consumption, CPU turbo boost mode is turned On

Figure 81 shows, in perspective, FPGA and CPU power consumed while performing Black-Scholes computations. FPGA consumes 8.96 times less power. However we still need to know how this power corresponds to the consumed energy. Theoretically a higher power device may use less energy if it executes its processing in less time.

To determine this “energy consumed” quality we use Performance per Watt as the criteria.

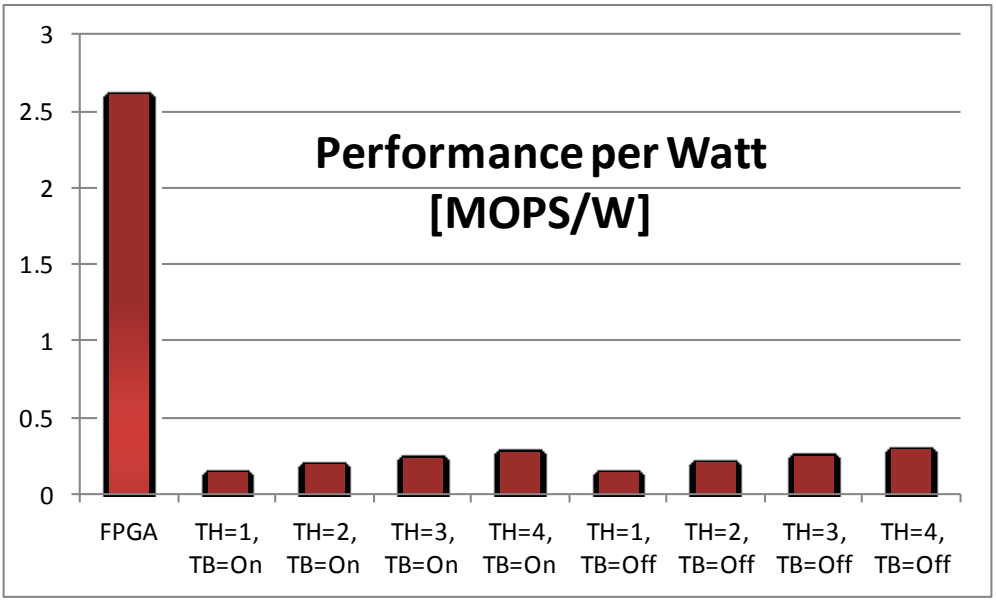

Figure 82 – FPGA and CPU Performance per Watt

Figure 82 shows Performance per Watt (PPW) for FPGA and CPU while performing Black-Scholes computations. “TH” stands for the number of threads (CPU cores) and “TB” indicates if CPU turbo boost mode is turned On or Off. FPGA offers 9.6 times better PPW when compared to a four-threaded program using the whole CPU. It is also important to note that FPGA PPW is shown based on the FPGA Achievable throughput. Any other throughput improvement would change this relation to be even more favorable towards the FPGA technology and suggested Electron architecture.

Chapter 7 - Conclusions and Future Work

7.1 Conclusions

Computer Clouds are major power consumers. It was reported that in 2010 global data centers accounted for 1.1% to 1.5% of total electricity use while in the US that number is between 1.7% and 2.2%.

FPGA technology offers better acceleration and lower power consumption than server CPUs. There are multiple ways of using FPGA to develop coprocessors. However if used in Cloud computing the solution space gets significantly smaller. Clouds service a multitude of applications therefore they require flexible hardware architectures as well as improved processing throughput. This flexibility comes from architectures that can do many things but they (a) do not require reconfiguration or (b) minimize cases when reconfiguration is required or (c) if reconfiguration is required minimize its scope and duration (partial reconfiguration). The process of reconfiguration and other design constraints as well as FPGA design methodologies applicable to Clouds are analysed and presented.

Currently available FPGA commercial and academic tools and architectures are reviewed. Some of them could meet a sub-set of criteria defined in this project for successful deployment in Clouds. However a more complete architecture is required.

Hardware architecture based on the Electron processor running nano-code and FPGA technology was proposed. This architecture is adaptable for a variety of computational problems. A case study is presented and advantages of the architecture are proven. This architecture meets all of the coprocessor Cloud deployment criteria.

7.2 Summary of Contributions

In this project it was identified how Cloud energy consumption can be reduced. FPGA technology is selected due to its low power consumption, speedup offered and flexibility.

Requirements for FPGA-based coprocessors are defined. Based on this set of criteria unique hardware architecture is developed. This architecture is FPGA-based with programmable control paths and controllable and reconfigurable data paths.

This architecture is constructed from the Electron processor and a controllable data path supported by a crossbar. Electron is a Very Long Control Word (VLCW) processor with a Code Stream unit supporting, among other things, branch instruction. A number of crossbar architectures are analysed based on offered throughput, logic resources and backend timing. Two types of crossbars are selected, one for Processing Tiles (2D mesh) and one for the Electron (sparse crossbar).

The Electron runs nano-code. The nano-code VLCW instruction format is defined. The code size compression technique is developed and implemented.

An example design was created based on 6 Processing Tiles each containing 4 Electron Processors. The Processing Tile system-level or Electron kernel nano-code could be built for parallel or sequential processing. In this project each of the 6 tiles is running the same program. The Electron kernel code uses the sequential model.

FPGA resource usage is analysed. The Electron processor and the crossbar take only 25% of the Electron sub-system. The remaining logic resources are taken by processing elements. Constraints for the correct estimate of processing element features (architectures, multi-threaded, number of instances) are defined.

System level and Kernel nano-code was developed for the Black-Scholes equation commonly used in Financial Industry.

Besides the Verilog RTL code, the verification test bench is developed to test the integrity of the RTL and nano-code.

Software implementation of the Black-Scholes equation running on a PC is developed. Software communication infrastructure consisting of a PCIe driver (design reuse) and test programs is developed. These programs are capable of write and read DMA transfers to the FPGA board running Black-Scholes nano-code.

Experiments were carried out on and Electron based design proved to be capable of competing with a CPU software based solution running on a PC. Measured results showed that for the Virtex 6 (XC6VLX240T) FPGA Electron based solution performance is more stable (smaller jitter), predictable and slightly faster than the i5-3570 CPU software based solution while offering 9.6x improved Performance per Watt .

It is possible to design software tools, such as the nano-code compiler. Required tool design methods and constraints are specified. The hardware architecture based on the Electron supports virtual hardware via pre-designed templates. This virtual hardware model provides unlimited flexibility therefore meeting the needs of many applications being run on Clouds.

Therefore Electron could find its use in the Cloud. It meets all criteria identified in this thesis for successful deployment in the Cloud Computing Environment.

7.3 Future Work

The following are examples of how this work may be continued.

7.3.1 Low latency PCIe Core

Section 6.8 indicated opportunities to improve the system throughput coming from PCIe cores. There are two PCIe cores used in this project: Xilinx low level PCIe core; and SpeedyPCIe core with the associated software driver. The SpeedyPCIe core offers DMA capabilities. It is also designed with consideration for future Xilinx FPGA families and with new and faster PCIe generations. Its design generalization does not offer the best performance, such as low latency.

A new core whose main design constraint is low latency is an excellent research area. This core should also offer compile time configuration capabilities which provide the exact number of interfaces as there are Processing Tiles on the FPGA.

Additionally the software driver that would go with this core should be designed to offer non-blocking write/read capabilities.

7.3.2 Monte Carlo Analysis

One of the computing applications used in the Financial Industry is regression testing (Section 6.6). A number of calculations are performed in an overnight run trying to emulate market conditions that took place a day before. A result of these regression test cases is a set of parameters used to fine tune equations used the next day during automatic security and option trading.

The Monte Carlo analysis relies on pseudo number generation. Pseudo random number generation is done faster by hardware than by software. Electron based FPGA solution would contribute to regression analysis because of:

- Inherited Electron benefits such as parallel processing,
- Faster pseudo number generation required for Monte Carlo and
- Low PC-FPGA communication overhead required for regression testing

The above described processing acceleration plus the lower electricity consumption provide benefits over the software based CPU regression solution.

7.3.3 Nano-code Compilation

In this project nano-code compilation was done by hand (Section 5.2.5). However documented steps could be used for automatic code compilation.

There is a huge body of research dedicated to graph theory. This research is used in code compilation to underlining hardware. However, a simple alternative method could also be used. In this method the computation problem is broken into pseudo-code based on equation parentheses and operation dependencies. Therefore in simple form a compiler could become a parsing script. The offloaded calculations would become the main program executed function calls. The functions would execute on the FPGA by the Electron.

7.3.4 Processing Elements Optimization

It was shown in Section 5.2 that Kernel 2 takes 136 clocks out of which 32 clocks are taken by the division operation (23% of the kernel processing time). Significant improvements could be achieved by custom designed commonly used processing elements (multiply-add or even for fundamental operations such as division is). As a consequence these processing

elements would grow in size. This could be remedied by using one such processing element per PT. This technique was suggested in Section 5.4.4.

A library of such modules in a form of pre-synthesised hardware modules could be created (virtual hardware), see Section 2.3.3.

7.3.5 Tools and Algorithms for resource estimation and template matching

It was shown in Section 5.4.3 that Electron Processing Elements could take 72% of Electron resources. Processing Elements (PE) are major contributors to processing throughput and Processing Tile size. Therefore the selection of Processing Elements and their features is an important task.

In this context a template is a member of a hardware architecture library (Section 2.3.3). The library contains pre-synthesized PEs representing a Virtual Hardware model.

A tool that parses application code to be offloaded and accelerated would determine PXB and PE requirements. Performance comparison estimation of the existing PEs (already available on the FPGA) and all the template library elements is carried out. This comparison takes into consideration reconfiguration speed, as well as size and the time of the computational problem. If a sufficiently faster solution could be obtained by replacing the existing PXB and/or PEs with a different template, FPGA would be partially reconfigured with new PEs.

Once the above steps are completed nano-code for existing hardware is generated and downloaded to Electron's code memory.

References

[1] M. Mills, "The Cloud Begins with Coal," 8 2013. [Online]. Available: http://www.tech-pundit.com/wp-content/uploads/2013/07/Cloud_Begins_With_Coal.pdf?c761ac. [Accessed 17 11 2013].

[2] A. Renbi, L. Lindh and J. Delsing, "Non-Instruction Fetch-Based Architecture Reduces Almost 100 Percent of the Dynamic Power and Energy," in *IEEE/ACM Int'l Conference on & Int'l Conference on Cyber, Physical and Social Computing (CPSCom) Green Computing and Communications (GreenCom)*, Hangzhou, 2010.

[3] A. Madhavapeddy and S. Singh, "Reconfigurable Data Processing for Clouds," in *IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Salt Lake City, UT , 2011.

[4] S. Weston, J.-T. Marin, J. Spooner, O. Pell and O. Mencer, "Accelerating the computation of portfolios of tranched credit derivatives," in *IEEE Workshop High Performance Computational Finance (WHPCF)*, New Orleans, LA, USA , 2010.

[5] J. G. Koomey, "Growth In Data Center Electricity Use 2005 To 2010," 2011. [Online]. Available: http://www.koomey.com/research.html. [Accessed 26 11 2013].

[6] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 4th ed., Morgan Kaufmann, 2007.

[7] W. Wang, M. Bolic and J. Parri, "pvFPGA: Accessing an FPGA-based hardware accelerator in a paravirtualized environment," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Montreal, QC, 2013.

[8] Academia.edu Share Research, "http://www.academia.edu/760613/Survey_of_Virtual_Machine_Migration_Techniques," Academia, [Online]. Available: http://www.academia.edu/760613/Survey_of_Virtual_Machine_Migration_Techniques. [Accessed 28 12 2013].

[9] D. Aikema, A. Mirtchovski, C. Kiddle and R. Simmonds, "Green cloud VM migration: Power use analysis," in *International Green Computing Conference (IGCC)*, San Jose, CA , 2012.

[10] H. Jin, L. Deng, S. Wu, X. Shi and X. Pan, "Live virtual machine migration with adaptive, memory compression," in *IEEE International Conference on Cluster Computing and Workshops*, New Orleans, LA , 2009.

[11] S. Chey, J. Liz, J. W. Sheaffery, K. Skadrony and J. Lachz, "Accelerating Compute-Intensive Applications with GPUs and FPGAs," in *Symposium on Application Specific Processors*, Anaheim, CA , 2008.

[12] D. B. Thomas, "Acceleration of financial Monte-Carlo simulations using FPGAs," *IEEE Workshop on High Performance Computational Finance (WHPCF),* 2010.

[13] D. B. Thomas, W. Luk and G. W. Morris, "FPGA accelerated low-latency market data feed processing," *17th IEEE Symposium on High Performance Interconnec,* 2009.

[14] A. Irturk, B. Benson, N. Laptev and R. Kastner, "FPGA Acceleration of Mean Variance Framework for Optimal Asset Allocation," in *Workshop on High Performance Computational Finance*, Austin, TX , 2009.

[15] B. Mackin and N. Woods, "FPGA Acceleration in HPC," Xtreme Data, 2006. [Online]. Available: http://www.hypertransport.org/docs/wp/FPGA_Acceleration_in_HPC_Nov06.pdf. [Accessed 10 11 2013].

[16] M. Chlistalla, "High Frequency Trading," Deutsche Bank Research, 7 February 2011. [Online]. Available: http://www.dbresearch.com/PROD/DBR_INTERNET_EN-PROD/PROD0000000000269468.pdf. [Accessed 10 11 2013].

[17] C. Leber, B. Geib and H. Litz, "High Frequency Trading Acceleration using FPGAs," in *International Conference on Field Programmable Logic and Applications (FPL)*, Chania, 2011.

[18] R. Pottathuparambil, J. Coyne, J. Allred, W. Lynch and V. Natoli, "Low-Latency FPGA Based Financial Data Feed Handler," in *FCCM '11 Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines* , Washington, DC, USA, 2011.

[19] C. Huang, F. Vahid and T. Givargis, "A Custom FPGA Processor for Physical Model Ordinary Differential Equation Solving," *IEEE Embedded Systems Letters,* vol. 3, no. 4, pp. 113 - 116, 2011.

[20] R. Weber, A. Gothandaraman, R. Hinde and G. Peterson, "Comparing Hardware Accelerators in Scientific Applications: A Case Study," *IEEE Transactions on Parallel and Distributed Systems,* vol. 22, no. 1, pp. 58-68, 2011.

[21] M. Lin, S. Cheng and J. Wawrzynek, "Cascading Deep Pipelines to Achieve High Throughput in Numerical Reduction Operations," in *International Conference on Reconfigurable Computing*, Quintana Roo , 2010.

[22] J. Lee, L. Shannon, M. Yedlin and G. Margrave, "A multi-FPGA application-specific architecture for accelerating a floating point Fourier Integral Operator," in *International Conference on Application-Specific Systems, Architectures and Processors, 2008*, Leuven , 2008.

[23] G. Falcao, M. Owaida, D. Novo, M. Purnaprajna, N. Bellas, C. D. Antonopoulos, G. Karakonstantis, A. Burg and P. Ienne, "Shortening design time through multiplatform simulations with a portable OpenCL golden-model: the LDPC

decoder case," in *IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, Toronto, ON , 2012.

[24] M. Owaida, N. Bellas, K. Daloukas and C. Antonopoulos, "Synthesis of Platform Architectures from OpenCL Programs," in *IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Salt Lake City, UT, 2011.

[25] N. Kapre and A. DeHon, "Accelerating SPICE Model-Evaluation using FPGAs," in *17th IEEE Symposium on Field Programmable Custom Computing Machines*, Napa, CA , 2009.

[26] Altera, "Implementing FPGA Design with the OpenCL Standard," November 2013. [Online]. Available: http://www.altera.com/literature/wp/wp-01173-opencl.pdf. [Accessed 26 11 2013].

[27] P. McLellan, "EDPS: SoC FPGAs," SemiWiki, 04 September 2012. [Online]. Available: http://www.semiwiki.com/forum/content/1155-edps-soc-fpgas.html#comments. [Accessed 11 November 2013].

[28] B. Betkaoui, D. Thomas and W. Luk, "Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing," in *International Conference on Field-Programmable Technology (FPT)*, Beijing, 2010.

[29] Steve Casselman, DRC Computer Corp, "EE Times," 25 7 2007. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1274588. [Accessed 10 11 2013].

[30] Wikipedia, "Wikipedia," Wikipedia, 10 11 2013. [Online]. Available: http://en.wikipedia.org/wiki/Cloud_computing. [Accessed 10 11 2013].

[31] Wikinvest, "Wikinvest," Wikinvest, [Online]. Available: http://www.wikinvest.com/concept/Cloud_Computing. [Accessed 10 11 2013].

[32] P. Rid, "Technobuffalo," 17 1 2010. [Online]. Available: http://www.technobuffalo.com/internet/five-examples-of-cloud-computing/. [Accessed 10 11 2013].

[33] I. G. B. S. E. Report, "The power of cloud Driving business model innovation," IT World, 2012.

[34] S. Kestur, J. Davis and O. Williams, "BLAS Comparison on FPGA, CPU and GPU," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2010*, Lixouri, Kefalonia , 2010.

[35] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Computing Surveys,* vol. 34, no. 2, p. 171–210, 2002.

[36] JP Morgan Chase, "FPGA Accelerators at JP Morgan Chase," 13 7 2011. [Online]. Available: http://www.youtube.com/watch?v=9NqX1ETADn0. [Accessed 11 November 2013].

[37] D. Chen and D. Singh, "Using OpenCL to evaluate the efficiency of CPUS, GPUS and FPGAS for information filtering," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, Oslo, 2012.

[38] B. Betkaoui, D. Thomas and W. Luk, "Comparing Performance and Energy Efficiency of FPGAs and GPUs for High Productivity Computing," in *International Conference on Field-Programmable Technology (FPT)*, Beijing, 2010.

[39] S. Skalicky, S. Lopez, M. Łukowiak, J. Letendre and D. Gasser, "Linear algebra computations in heterogeneous systems," in *IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, Washington, DC , 2013.

[40] Intel, "Heterogeneous Computing in the Cloud: Crunching Big Data and Democratizing HPC Access for the Life Sciences," 12 July 2012. [Online]. Available:

http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/xeon-phi-life-sciences-computing-paper.pdf. [Accessed 5 1 2014].

[41] M. Stojilovic, D. Novo, L. Saranovac, P. Brisk and P. Ienne, "Selective flexibility: Breaking the rigidity of datapath merging," *Design, Automation & Test in Europe Conference & Exhibition,* 2012.

[42] M. Lin, I. Lebedev and J. Wawrzynek, "OpenRCL: Low-Power High-Performance Computing with Reconfigurable Devices," in *International Conference on Field Programmable Logic and Applications (FPL)*, Milano , 2010.

[43] V. Groza and R. Abielmona, "What next? A hardware operating system?," in *Proceedings of the 21st IEEE Instrumentation and Measurement Technology Conference*, 2004.

[44] B. Holland, M. Vacas, V. Aggarwal, R. DeVille, I. Troxel and A. D. George, "Survey of C-based Application Mapping Tools for Reconfigurable Computing," 2005. [Online]. Available: https://wiki.ittc.ku.edu/eecs700/images/b/b8/Holland.pdf. [Accessed 30 11 2013].

[45] P. Karlström, Z. Wenbiao, W. Chinghan and D. Liu, "Design of PIONEER: A case study using NoGap," in *2010 Asia Pacific Conference on Postgraduate Research in Microelectronics and Electronics (PrimeAsia),*, Shanghai, 2010.

[46] A. Kritikakou, F. Catthoor, G. Athanasiou, V. Kelefouras and C. Goutis, "A template-based methodology for efficient microprocessor and FPGA accelerator co-design," in *International Conference on Embedded Computer System*, Samos, 2012.

[47] A. Nanda, "Accelerate Performance and Design Productivity with OpenCL on Altera FPGAs," Altera, 2012. [Online]. Available: http://www.altera.com/education/webcasts/all/source-files/wc-2012-opencl/player.html?GSA_pos=2&WT.oss_r=1&WT.oss=opencl. [Accessed 16 11

2013].

[48] SRC Computers, "SRC Technology," SRC Computers, [Online]. Available: http://www.srccomp.com/techpubs/techoverview.asp. [Accessed 10 11 2013].

[49] Forte Design Systems, "Cynthesizer 5," Forte Design Systems, [Online]. Available: http://www.forteds.com/products/cynthesizer.asp. [Accessed 10 11 2013].

[50] Mentor Graphics, "Handel-C Synthesis Methodology," Mentor Graphics, [Online]. Available: http://www.mentor.com/products/fpga/handel-c/. [Accessed 10 11 2013].

[51] Impulse Accelerated, "Impulse CoDeveloper C-to-FPGA Tools," impulseaccelerated, [Online]. Available: http://www.impulseaccelerated.com/products_universal.htm. [Accessed 10 11 2013].

[52] R. Goering, "Open-source C compiler targets FPGAs," EE Times, 18 November 2002. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1200189. [Accessed 10 11 2013].

[53] M. Gokhale, J. Frigo, C. Ahrens and M. Popkin, "Sc2 C-to-FPGA Compiler," MIT, [Online]. Available: http://www.ll.mit.edu/HPEC/agendas/proc02/presentations/pdfs/gokhale.PDF. [Accessed 10 11 2013].

[54] D. Pellerin, Practical FPGA Programming in C, 1st ed., Prentice Hall, 2005.

[55] O. Mencer, "ASC: A Stream Compiler for Computing With FPGAs," *IEEE Transactions on computer-aided design of integrated circuits and systems,* vol. 25, no. 9, pp. 1603 - 1617 , September 2006.

[56] Maxeler, "Maximum Performance Computing," Maxeler, [Online]. Available: http://www.maxeler.com. [Accessed 10 11 2013].

[57] Berkeley Design Technology, Inc, "High-Level Synthesis Tools for Xilinx FPGAs," 2010. [Online]. Available: http://www.xilinx.com/technology/dsp/BDTI_techpaper.pdf. [Accessed 10 11 2013].

[58] D. Navarro, O. Lucia, L. Barragan, I. Urriza and O. Jimenez, "High-Level Synthesis for Accelerating the FPGA Implementation of Computationally Demanding," *IEEE Transactions on Industrial Informatics,* vol. 9, no. 3, pp. 1371 - 1379 , 2013.

[59] J. Hiraiwa and H. Amano, "An FPGA Implementation of Reconfigurable Real-Time Vision Architecture," in *27th International Conference on Advanced Information Networking and Applications*, Barcelona, 2013.

[60] A. W. Bohm, "Compiling high-level programs to FPGA configurations," University, Colorado State, [Online]. Available: http://www.cs.colostate.edu/cameron/index.html. [Accessed 10 11 2013].

[61] N. Rotem, "Compile Your C code into Verilog," 2009. [Online]. Available: http://www.c-to-verilog.com/. [Accessed 10 11 2013].

[62] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong and W.-M. Hwu, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *IEEE 7th Symposium on Application Specific Processors*, San Francisco, CA , 2009.

[63] J. Curreri, S. Koehler, B. Holland and A. George, "Performance Analysis with High-Level Languages for High-Performance Reconfigurable Computing," in *16th International Symposium on Field-Programmable Custom Computing Machines*, Palo Alto, CA , 2008.

[64] D. Sanchez-Roman, G. Sutter, S. Lopez-Buedo, I. Gonzalez, F. Gomez-Arribas, J. Aracil and F. Palacios, "High-Level Languages and Floating-Point Arithmetic for FPGA-Based CFD Simulations," in *IEEE Design & Test of Computers*, 2011.

[65] D. Sanchez-Roman, G. Sutter, S. Lopez-Buedo, I. Gonzalez, F. Gomez-Arribas and J. Aracil, "An Euler solver accelerator in FPGA for computational fluid dynamics applications," in *VII Southern Conference on Programmable Logic (SPL)*, 2011.

[66] D. Gajski and M. Reshadi, "A cycle-accurate compilation algorithm for custom pipelined datapaths," in *Hardware/Software Codesign and System Synthesis*, Jersey City, NJ, USA , 2005.

[67] D. Ivosevic and V. Sruk, "Evaluation of embedded processor based BDD implementation," in *Proceedings of the 33rd International Convention MIPRO, 2010 Proceedings of the 33rd International Convention*, Opatija, Croatia , 2010.

[68] B. Gorjiara and D. Gajski, "Automatic architecture refinement techniques for customizing processing elements," in *Design Automation Conference*, Anaheim, CA , 2008.

[69] B. Gorjiara and D. Gajski, "Custom processor design using NISC: a case-study on DCT algorithm," *Embedded Systems for Real-Time Multimedia,* pp. 55 - 60 , 2005.

[70] M. Reshadi, B. Gorjiara and D. Gajski, "Utilizing horizontal and vertical parallelism with a no-instruction-set compiler for custom datapaths," in *2005 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 2005.

[71] B. Gorjiara, M. Reshadi and D. Gajski, "Designing a custom architecture for DCT using NISC technology," in *Asia and South Pacific Conference on Design Automation*, Yokohama, 2006.

[72] M. Reshadi, B. Gorjara and D. Gajski, "C-Based Design Flow: A Case Study on G.729A for Voice over Internet Protocol (VoIP)," in *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE* , Anaheim, CA , 2008.

[73] D. Ivosevic and V. Sruk, "Automated modeling of custom processors for DCT

algorithm," in *Proceedings of the 34th International Convention MIPRO*, Opatija, Croatia , 2011.

[74] J. Trajkovic and D. Gajski, "Custom Processor Core Construction from C Code," in *Symposium on Application Specific Processors, 2008. SASP 2008.*, Anaheim, CA , 2008.

[75] B. Gorjiara, "Controlling LEDs and Switches on a Xilinx Vertix-4 Video Starter Kit Using NIOS COmponents," Donald Bren School of Information and Computer Sciences, [Online]. Available: http://www.ics.uci.edu/~nisc/designs/XilinxBoard_VideoStarterKit/LEDs-Switches/LEDs-and-Switches.pdf. [Accessed 10 11 2013].

[76] M. Thuresson, M. Sjalander, M. Bjork, L. Svensson, P. Larsson-Edefors and P. Stenstrom, "FlexCore: Utilizing Exposed Datapath Control for Efficient Computing," in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, IC-SAMOS 2007.*, Samos , 2007.

[77] S. Hong, Sadasivam and X. L. M., "Post-generation of overall execution controller for data centric signal processing algorithms," *Proceedings 7th International Conference on Signal Processing,* vol. 1, pp. 551 - 554, 2004.

[78] X. Liang, A. Athalye and S. Hong, "Dynamic coarse grain dataflow reconfiguration technique for real-time systems design," *IEEE International Symposium on Circuits and Systems,* vol. 4, pp. 3511 - 3514, 2005.

[79] X. Liang, A. Athalye and S. Hong, "Equalizing data-path for processing speed determination in block level pipelining," *IEEE International Symposium on Circuits and Systems,* vol. 2, pp. 1646 - 1649, 2005.

[80] S. Hong, J. Lee, A. Athalye, P. Djuric and W.-D. Cho, "Design Methodology for Domain Specific Parameterizable Particle Filter Realizations," *IEEE Transactions on Circuits*

and Systems, vol. 54, no. 9, pp. 1987 - 2000 , 2007.

[81] W. Chun, S. Yoon and S. Hong, "Buffer Controller-Based Multiple Processing Element Utilization for Dataflow Synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 19, no. 7, pp. 1249 - 1262, 2011.

[82] H. Jung, K. Lee and S. Ha, "Efficient hardware controller synthesis for synchronous dataflow graph in system level design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 10, no. 4, pp. 423 - 428 , 2002.

[83] D. A. Patterson and J. L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, Morgan Kaufmann, 2012.

[84] S.-T. Chuang, A. Goel, N. McKeown and B. Prabhakar, "Matching output queueing with a combined input output queued switch," in *Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, New York, NY , 1999.

[85] Y. Zheng and C. Shao, "An Efficient Round-Robin Algorithm for Combined Input-Crosspoint-Queued Switches," in *Joint International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services*, Papeete, Tahiti , 2005.

[86] T. Javidi, R. Magill and T. Hrabik, "A High Throughput Scheduling Algorithm for a Buffered Crossbar Switch Fabric," *IEEE International Conference on Communications,* vol. 5, pp. 1586 - 1591, 2001.

[87] S.-T. Chuang, S. Iyer and N. McKeown, "Practical algorithms for performance guarantees in buffered crossbars," *Proceedings IEEE INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. ,* vol. 2, pp. 981 - 991, 2005.

[88] M. Lin and N. McKeown, "The Throughput of a Buffered Crossbar Switch," *IEEE*

Communications Letters, vol. 9, no. 5, pp. 465 - 467 , 2005.

[89] J. Turner, "Strong Performance Guarantees for Asynchronous Buffered Crossbar," *IEEE/ACM Transactions on Networking,* vol. 17, no. 4, pp. 1017 - 1028 , 2009.

[90] M. Modarressi, A. Tavakkol and H. Sarbazi-Azad, "Application-Aware Topology Reconfiguration for On-Chip Networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 19, no. 11, pp. 2010 - 2022 , November 2011.

[91] M. Stensgaard and J. Lyngby Sparso, "ReNoC: A Network-on-Chip Architecture with Reconfigurable Topology," *Second ACM/IEEE International Symposium on Networks-on-Chip, 2008. NoCS 2008,* pp. 55-64, April 2008.

[92] D. Greaves and S. Singh, "Distributing C# methods and threads over Ethernet-connected FPGAs using Kiwi," in *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, Cambridge, 2011.

[93] J. A. Fisher, "Very Long Instruction Word Architectures And The Eli-512," *Solid-State Circuits Magazine, IEEE,* vol. 1, no. 2, pp. 23-33, 1988.

[94] S. Rixner, Stream Processor Architecture, 1st ed., Springer, 2001.

[95] IBM, PowerLinuxTeam, "Practical experiences with OS Jitter," IBM, PowerLinuxTeam, April 2013. [Online]. Available: http://www.ibm.com/developerworks/. [Accessed 10 11 2013].

[96] Altera Corporation , "Reducing Power Consumption and Increasing Bandwidth on 28-nm FPGAs," March 2012. [Online]. Available: http://www.altera.com/literature/wp/wp-01148-stxv-power-consumption.pdf. [Accessed 13 11 2013].

[97] Wikipedia, "Amdahl's Law," Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Amdahl%27s_law. [Accessed 27 December 2013].

[98] M. Hill and M. Marty, "Amdahl's Law in the Multicore Era," *Computer,* vol. 41, no. 7, pp. 33-38, 2008.

[99] RiskEncyclopedia, "Merton (1973) Option Pricing Formula," RiskEncyclopedia, [Online]. Available: http://riskencyclopedia.com/articles/merton_1973/. [Accessed 10 11 2013].

[100] Wikipedia, "Numerical approximations for the normal CDF," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Normal_distribution. [Accessed 10 11 2013].

[101] R. Bittner, "Speedy bus mastering PCI express," in *International Conference on Field Programmable Logic and Applications*, Oslo, 2012.

[102] R. Bittner, "Speedy PCIe User Guide v1.0," 2012. [Online]. Available: http://www.google.ca/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&ved=0CCsQFjAA&url=http%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F192884%2F20130513%2520Cluster%2520Computing%2520Journal%2520Springer%2520Final%252010.1007_s10586-013-0280-9.pdf&ei=de6bUo. [Accessed 01 12 2013].

[103] D. B. Thomas and W. Luk, "An FPGA-Specific Algorithm for Direct Generation of Multi-Variate Gaussian Random Numbers," in *21st IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP)*, Rennes, France , 2010.

[104] A. Tse, D. Thomas, K. Tsoi and W. Luk, "Dynamic scheduling Monte-Carlo framework for multi-accelerator heterogeneous clusters," in *International Conference on Field-Programmable Technology (FPT)*, Beijing , 2010.

[105] I. Godard, "Drinking from the Firehose: How the Mill CPU Decodes 30+ Instructions per Cycle," Stanford Seminar, 4 6 2013. [Online]. Available:

http://www.youtube.com/watch?v=LgLNyMAi-0I. [Accessed 20 11 2013].

[106] S. Keckler, W. Dally, D. Maskit, N. Carter, A. Chang and W. Lee, "Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor," in *The 25th Annual International Symposium on Computer Architecture*, Barcelona, 1998.

[107] Wikipedia, "Microcode," Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Microcode. [Accessed 27 12 2013].

[108] K. Ovtcharov, I. Tili and J. Steffan, "A Multithreaded VLIW Soft Processor Family," in *IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Seattle, WA , 2013.

Appendix A – Speedy PCIe

In this section some Speedy PCIe operation specific are documented.

Slave Write

Bus master initiates a slave write PCIe transfer with 24x 16-DWs totaling to 1536-bytes. 16-DWs transfer is the smallest unit of transfer available from Speedy PCIe (driven by the Speedy PCIe architecture).

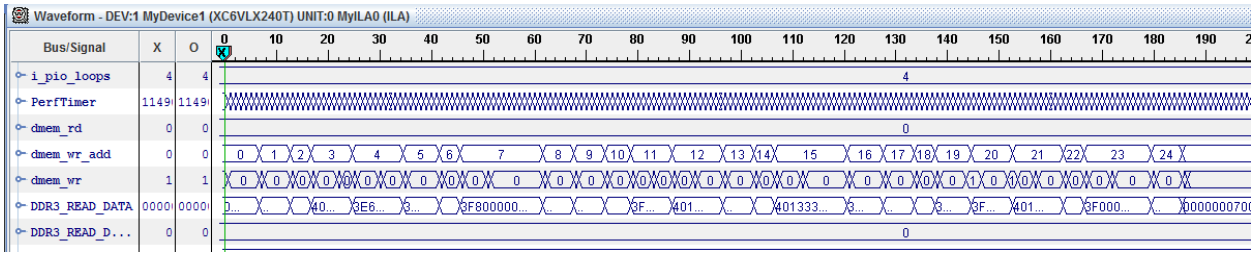

Figure 83 – Slave Write - Waveform

Slave Read

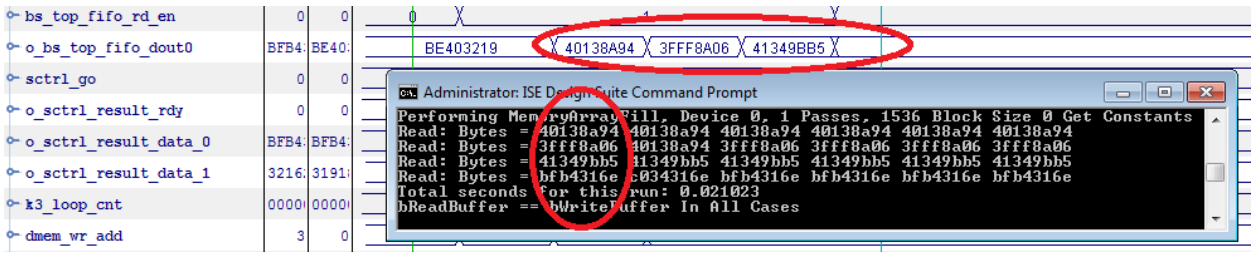

Figure 84 – Slave Read - Waveform

Slave Read – Speedy PCIe extra read

Speedy PCIe works very well. However there is an issue a user needs to be aware of when the FPGA IP or software driver creates an extra read under some conditions. This extra read cannot be provided by Processing Tiles since it is a read for non-existing data

therefore this extra read cannot be completed. This would cause the whole application to be blocked due to the software driver waiting for this extra read to complete.

Here is an example. A slave read request for 512-DWs was placed. There are 32 “expected” reads (32x 16-DWs totaling to 2K). Approximately 125 clock cycles (5 nS clock period) after the expected reads there is an extra read request. It never gets completed since there is nothing to read.

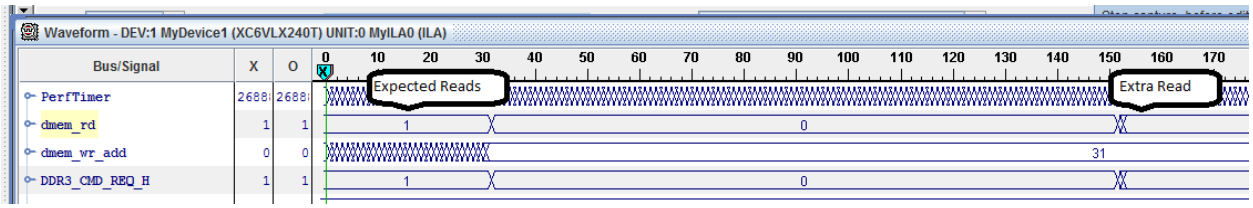

Figure 85 – Speedy PCIe 32x 16-DW slave read

This unconfirmed Speedy PCIe “bug” was addressed by implementing a workaround within the FPGA. If an extra slave read were requested for Processing Tile results that are never calculated by FPGA, FPGA would respond with 32’hBEEF_BEEF data.

There is approximately 600 nS penalty on top of expected slave read latency – affecting to some degree some of Processing Tile throughput results.

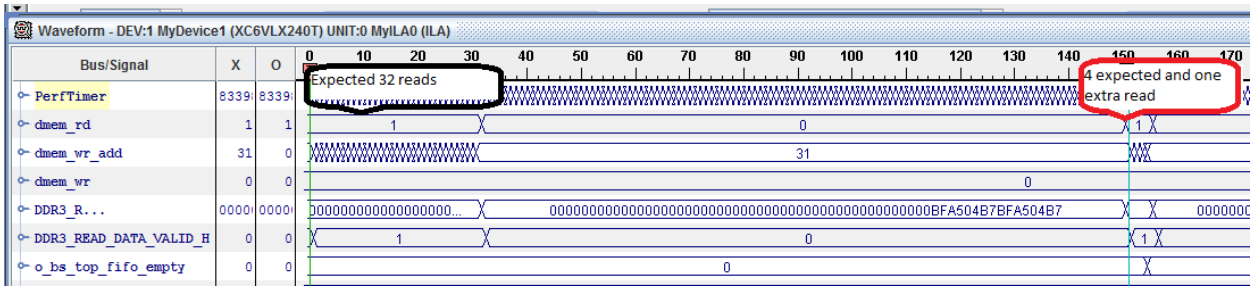

Figure 86 – Speedy PCIe 36x 16-DW slave read

For some reasons, and not always extra write would take place. DDR3_CMD_DATA_MASK_L, if low, indicates that data are valid. Two red ovals depict two moments where the same data is written 2 times on the same address. This behaviour makes use of a FIFO inconvenient since addresses need to be compared (previous with the new) to ensure no double writes take place.

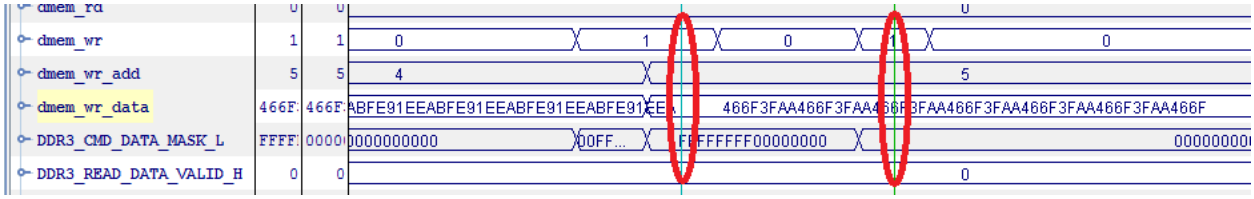

Figure 87 – Speedy PCIe Bug