High throughput “on-the-fly” Stream Data Processing on hybrid FPGA-Based platform

Rousopoulos Christos

Supervisor: **Prof. Apostolos Dollas**
Prof. Dionisios Pnevmatikatos
Asoc. Prof. Ioannis Papaefstathiou

School of Electrical and Computer Engineering (ECE)
Technical University of Crete

This dissertation is submitted for the degree of
Master of Science

July 2017

I would like to dedicate this thesis to my loving parents ...

Acknowledgements

I would like to thank my supervisor Professor Apostolos Dollas for his trust, continuous support and guidance throughout the years of this master degree. And I also want to apologize to him for “trubles” I put him. Also, i would like to thank Professor Dionysios Pnevmatikatos and Prof. Ioannis Papaefstathiou for their interest in my thesis and their participation in the examination committee.

Also I would like to thank all the MHL members for the beautiful moments we had together in the lab.

I am eternally grateful to my friend Gregory Chrysos for his endless support in the fulfillment of this thesis. He is a rare person. I am really thankful to Evripides Sotiriadis for his valuable advices.

I would also like to thank my friend Thanasis Stratikopoulos for his continuous support.

I would like to thank all my really close friends for all the unforgettable years we had in Chania. I am thankful to my dear Tessy Skodra for encouraging and supporting me unconditionally, in the last few years.

Last but not least, I would like to thank my parents Kostas and Panagiota and my sisters Vagia and Niki, who encouraged me to this attempt and mainly who supported and continue to support, by all means, me and my choices. A big thank is not enough to express my gratitude to them! This work is dedicated to them.

Abstract

With the dawn of social networks and the growth of the Internet of Things market, huge volumes of data are generated every second in various fields, such as banking, agriculture, finance, health care, bioinformatics etc. This data congestion rises the need for systems that offer real-time fast processing of this high-volume and velocity data, while looking for low-cost and power efficient solutions. The stream join is a fundamental and computationally expensive operation for relating information from different data streams, usually applied over a specific time-based window due to the unbounded size of the data streams. This thesis presents an innovative high performance FPGA-based system for accelerating stream join processing.

The implemented system achieves at least one order of magnitude better processing throughput in comparison to other state-of-the-art software-based solutions and totally outperforms all lately proposed hardware-based solutions. Finally, the proposed hardware-based architecture is fully configurable and can be used as generic template for mapping stream processing algorithms on reconfigurable logic, taking into consideration real-world challenges and restrictions.

Table of contents

List of figures

List of tables

Nomenclature

Acronyms / Abbreviations

API Application Programming Interface

CLB configurable logic block

DSP Digital Signal Processing

FPGA Field-Programmable Gate Array

MIMD Multiple Instruction Multiple Data

MIMD Multiple Instruction Multiple Data

PTs Processing Threads

PU Processing Unit

SPE Stream Processor Engine

SPU Stream Processor Unit

Chapter 1

Introduction

The need for efficient real-time data analytics is an integral part of a growing number of data management technologies such as intrusion detection, High-frequency trading, and (complex) event processing. What is common among all these scenarios is a predefined set of continuous queries and unbounded event streams of incoming data that must be processed against the queries in real-time or almost real time. The challenges for today's real-time data analytics platforms are to meet the ever growing demands in processing large volumes of data at predictably low latencies across many application scenarios. The volume of traffic on the Internet has undergone an immense increase over the last decade. While according to Gilbert's law, communication bandwidth is projected to double every 9 to 10 months, conventional computation system architectures are showing signs of saturation in terms of offering the necessary processing power to sustain demands imposed by future Internet bandwidth growths.

The need for more processing power is the key ingredient in enabling innovation in high-throughput real-time data analytics to process, analyze, and extract relevant information from streams of events. Therefore, as proliferation of data and bandwidth continues, it is becoming essential to go beyond the conventional software-based approaches and adopt other key enabling technologies such as reconfigurable hardware in form of Field Programmable Gate Arrays (FPGAs). An FPGA is a cost-effective hardware acceleration solution that has the potential to excel at analytics-based computations due to its inherent parallelism. FPGAs can exploit low-level data and functional parallelism in applications with custom, application-specific circuits that can be re-configured, even after the FPGA has been deployed. In addition, FPGAs can meet the required elasticity in scaling out to meet increasing throughput demands. Furthermore FPGAs in combination with the recently appeared sophisticated memory systems offering high memory bandwidth contribute in faster and more efficient real-time stream data processing in Big Data areas.

1.1 Contribution

Stream data analysis is a high demanding area, in which many factors must be balanced in order to achieve the highest processing throughput and the lowest processing latency. Considering these requirements alongside with the flexibility that FPGAs offer, we designed a generic FPGA-based stream processing system capable of achieving orders of magnitude better throughput in comparison to the original software version and previous FPGA-based works. In particular, one of the most popular stream join operator algorithms ScaleJoin [7] was decided to be used as a use case study, aiming to investigate the capabilities of reconfigurable systems in many problems operating on streaming data.

In summary, the key contributions of this master thesis are the following:

- **The proposed design is a novel hardware architecture for stream data processing algorithms, and takes advantage of the fine and coarse grained parallelism on reconfigurable hardware.** The design is based on the processing strength of systolic arrays, which have been designed as a various sized array structure of tightly coupled data processing units on resulting on very deep pipeline. The size of this array is modular for the user, as it is possible to cascade more than one streaming processing modules, thereby achieving high performance on the available platform.

- **The final system allows the user to decide the number of the processing modules according to the desired performance requirements.** The best throughput on the targeted platform is performed, when the system fits as many streaming multi-processing modules as the available memory system ports.

- **This thesis presents the analysis of the achieved throughput for different configurations of streaming processing modules.**

- **The performance evaluation shows that the proposed systems can outperform by at least one order of magnitude any other state-of-the-art software-based or hardware-based solution.** The previous FPGA-based implementation run on the same target platform, whereas the software solution runs on a high-end multiprocessor platform. Therefore, results indicate the FPGAs as extremely appealing for real time data stream processing.

- **The final system is generic, as it is designed to support any algorithm within the stream processing family. Moreover, the design is scalable**, since it can be mapped on bigger FPGA systems, that can accommodate more streaming processing modules and more memory ports, leading to significantly higher throughput. Such systems

belong to the next generation platforms, which recently came up, and can exploit higher memory bandwidth.

1.2 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 provides a background of Big Data and Stream Data Processing, also a small introduction of FPGAs is provided. And finally, a brief description of the target platform used for developing and evaluating the designed system proposed in this thesis. Chapter 3 presents the up to now software and hardware based related work on stream processing and specifically stream join operator. Chapter 4 provides a deep analysis of ScaleJoin algorithm, which is the algorithm chosen to be mapped on reconfigurable logic. Also, in this chapter two known previous hardware accelerated versions of ScaleJoin are analyzed and compared, there benefits and drawbacks, which are the main motivation behind this work, are also presented. Chapter 5 presents the proposed system and also the main challenges with their solutions come up during it's development. Chapter 6 shows the experimental setup and the results of this work, while Chapter 7 suggests future improvements of this work and concludes the thesis.

Chapter 2

Background

The rise of the Internet in combination with the development of social networks and the Internet of Things (IoT) [14], have given the data more power. Huge amount of data is generated and collected every second in different applications in various fields as banking, agriculture, finance, health care, bioinformatics etc. [21] The fact that more than 2.5 quintillion bytes of data gets generated each day and that around 90% of the total amount of data present in this world has been created in the last two years indicates that the rate of data growth is extremely high and ever increasing [10]. This introduces the concept of "Big Data".

2.1 Big Data

Big Data is a general term given to massive quantities of information that are complex enough to demand innovative processing techniques to gain deep and valuable insights into the data.It is characterized by certain specific and complex features which make handling of big data unique and different from handling just large amounts of data. Big data has three essential features:

Volume: The amount of data is very large, usually in an order (such as TB and PB) that can not be stored and processed by a single machine. So, data need to be distributed stored in a cluster, then processed in a parallel and distributed way. Therefore, big data processing and cloud computing technologies are closely linked.

Variety: Data Variety refers to the fact that data is not restricted to be of the same type or structure. It may exist in different formats such as documents, images, emails, text, video, audio and other kinds of machine generated data such as from sensors, GPS signals, RFID tags, DNA analysis machines, etc. The data might follow a particular

structure or might be completely unstructured in nature. Variety also indicates that data may have to be captured, blended and aligned from different sources. This increases the complexity of the data analysis as not only it has to be cost efficient while retrieving different types of data, it also has to be designed in such a way that it is flexible and efficient enough to handle data from multiple sources.

Velocity: Generally, recent data is more valuable, so old data should be withdrawn when new data arrives. At the same time, the system needs to respond in a real-time or almost real-time manner.

Two new concepts that get introduced when Velocity is concerned are Latency and Throughput. Throughput describes the rate at which data is being consumed or ingested into the system for processing whereas Latency is the total time taken to process the data and generate the output since the data entered into the system. While trying to process or analyze data in motion, it is very important to keep up with the speed at which data is being ingested into the system. The architecture of the system capturing, analyzing and processing data streams should be able to support real-time turnaround time, that is, a fraction of a second. Also this level of efficiency must be consistent and maintained over the total period of processing time.

This characteristic led to the development of new technologies which focus on real-time processing of data streams. These technologies capture data from varied sources and of varied data types continuously in streams and quickly process them to generate output almost instantaneously without much time lag.

2.2 Issues with Big Data

While the potential benefits of big data is really significant, there remain many challenges that should be addressed. The most significant challenge is a result of the size of Big Data. The larger the data set to be processed, the longer it will take to analyze. Processing large and rapidly increasing amounts of data has been a target for many decades. In the past, this target was relying on the computation capacity of common processors. But now the data volume is scaling faster than the resources, and the capacity of CPU increases slowly. Thus, parallel and distributed solutions raised as alternatives, from Open MPI [15], to the famous Hadoop ecosystem [8] and Apache Spark [2]. To meet the increasingly growing large amount of data and to optimize the performance, the parallel and distributed computing platforms are also engaged in constantly upgrading. The second challenge is the cost of network communication

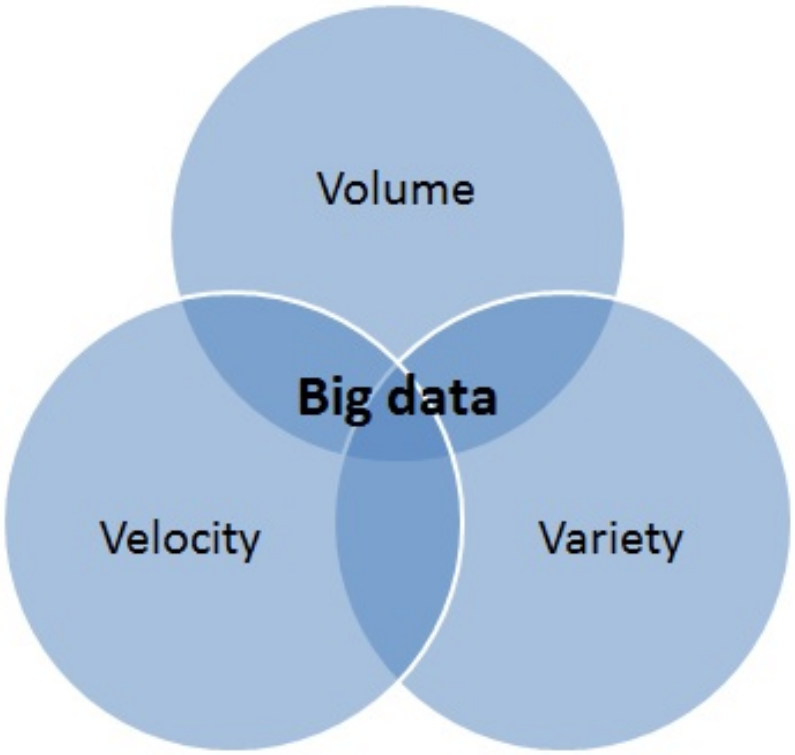


Fig. 2.1 A diagrammatic representation of the three essential Big Data features.

in transferring data. The transmission of data always became the bottleneck of a parallel platform. So when designing a method to compute an algorithm in parallel, minimizing the data transmission cost is a very important. The last but not least important issue comes from the dynamics of data. After the revolution of the Internet technologies and the popularity of social networks, data is now more dynamic. Users add new data to the network every second. And often, the latest data is the most valuable. This process can be described as a stream of data which constantly updates. Traditional distributed processing platforms process the data in batch. Each data update will trigger a re-calculation. This obviously can not meet the real-time processing requirement.

2.3 Real-Time processing for Data Streams

Recently made their appearance applications where the data is not processed as persistent static relations but rather as transient dynamic data streams. These applications include financial analysis, network monitoring, telecommunications data management, traffic data monitoring, web applications, manufacturing, sensor networks, and so on. In this kind of computational model, data items are formated by a record with a time-stamp. The data streams continuously arrive in multiple, rapid, time-varying, unpredictable and unbounded manners. For this type of data, batch-oriented data processing is not enough. Real-time query processing and in-stream processing is the immediate need in many applications. A data stream processing system needs to cope with the huge dynamicity in data streams in the near future, both at the architecture and the application level. At the architecture level it should be possible to add or remove computational nodes based on the current load. At the application level, it should be able to withdraw old results and take new coming data into account. However, current parallel processing systems for big data, being tailored to optimal realization of predefined analytics (static monitoring), are lacking the flexibility required for sensing and dynamic processing of changes in the data.

2.4 Stream Processing

In Big Data there is the need to process the data fast, so that the system can react to the changing conditions in real time. This requirement of processing high-volume data streams with low-latency becomes increasingly important in different applications in various fields, thus increasing the demand for stream processing. Under this requirement, the capacity of processing big volumes of data is not enough, we also need to react as fast as possible to the update of data. Stream processing is a programming model, which is also called

dataflow programming or reactive programming. A stream is a sequence of data and a series of operations will be applied to each element in the stream. Data items in streams are volatile, they are discarded after some time. Since stream processing often involves large amount of data, and requires the results in real-time, the stream processing platforms often process data in parallel. Moreover, this model is a good complement of parallel processing, and allows applications to more easily exploit a limited form of parallel processing. It simplifies parallel processing by restricting the parallel computation that can be performed. When data arrives in a stream or streams, data will be lost if it is not processed immediately or stored. Moreover, data usually arrives so rapidly that it is not feasible to store it all like in a conventional database, to process it when needed. So stream processing algorithms often rely on concise, approximate synopses of the input streams in real time computed with a simple pass over the streaming data. Below, we will first present the general principles of data stream processing and then introduce an approach to summarize a stream with only looking at fixed-length windows.

2.4.1 Data Stream processing Introduction

2.4.2 Sliding Window Model

One technique for producing results of a data stream query is to evaluate the query not over the entire past history of the data streams, but only over sliding windows of recent data from the incoming streams. Imposing sliding window model on data streams is a natural method for approximation. It is deterministic, so there is no danger that unfortunate random choices will produce a bad approximation. Most importantly, it emphasizes recent data, which in the majority of real-world applications is more important and relevant than old data: if one is trying in real-time to make sense of network traffic patterns, or phone call or transaction records, or scientific sensor data, then in general insights based on the recent past will be more informative and useful than insights based on stale data. In fact, for many such applications, sliding windows can be thought of not as an approximation technique reluctantly imposed due to the infeasibility of computing over all historical data, but rather as part of the desired query semantics explicitly expressed as part of the user's query [3].

2.4.3 Sliding Window Join

Sliding window joins have been widely studied. A sliding window join uses two or more streams of data as input, with window sizes for each stream. The output is also a stream,

containing all pairs of tuples from all incoming streams. An output pair is created only if the following criteria are satisfied.

1. all tuples of incoming streams satisfy the join predicate.
2. Each tuple of the pair is in the current active window of its respective stream.

Stream joins are only allowed when the state does not grow indefinitely. The join condition must ensure that a data of one input stream only joins with a bounded range of data from the other input streams. Having n data streams, and n corresponding sliding windows for each stream, the object of a sliding window join is to evaluate the join operation of the n windows. For each newly arrived data s, the join operation needs to probe all unexpired data and present in the sliding window at the arrival time of s, and return all the results that satisfy all the join predicates. And s will be kept in the window until it expires, and joined with other new coming data.

2.5 FPGAs

Traditional single threaded CPUs execute a sequential program and are unable to take advantage of the parallelism that exists in some applications, either data or task level parallelism. General purpose processors attempt to take advantage of parallelism in applications by utilizing numerous processing cores and partitioning the computation amongst those cores, as is done in multi-core processors or clusters. However, adding multiple cores for parallel processing requires the addition of more logic than the required computational unit, and that extra logic simply adds power and delay overhead to an application. Hardware systems can be designed to exploit the parallelism that exists in a problem in order to produce a much higher computational throughput as compared to a CPU. Hardware chips that are designed and fabricated to target a specific design problem are called application specific integrated circuits (ASICs). ASICs can perform computations in parallel, resulting in a great speedup over a CPU. However, ASICs require a large amount of cost in order to be produced, both for the design and fabrication of the chip. Field programmable gate arrays (FPGAs) can provide a middle ground, which proves very useful in some applications. FPGAs are a sea of reconfigurable logic and programmable routing that can be connected and configured to mirror the function of any digital circuit. The combinational logic in an FPGA is implemented using static random-access memory (SRAM) based lookup tables (LUTs), and the sequential logic is implemented using registers. They also contain additional features, such as digital signal processing (DSP) blocks and large memories with low latency access times. This allows FPGAs to perform massively parallel computation, giving them a

performance advantage over CPUs, but can also be reconfigured to fit multiple applications. The fine-grained parallelism in FPGAs gives them a performance advantage over traditional CPUs for applications requiring large amounts of computation on large data sets. The ability to be reconfigured gives them a reusability advantage over ASICs. However, both advantages are not without cost. Programming of FPGAs has traditionally been difficult because the designer has been required to write in a low-level hardware description language (HDL), such as Verilog or VHDL, instead of common and easier to debug high level software languages like C++. In spite of this, FPGAs can produce great improvements in both system runtime and power savings.

2.6 Convey HC-2ex Hybrid Platform

2.6.1 Introduction

The convey-computers introduced a hybrid platform that is based on reconfigurable logic. The convey HC-2ex is the second generation of the platform and combines an Intel Xeon processor and a Convey designed coprocessor based on Xilinx Field Programmable Gate Arrays, with its own high-bandwidth, virtual memory addressed and cache coherent memory subsystem. It also offers an ANSI standard development environment, increasing productivity and portability.

The coprocessor system Figure 2.2 is based on reconfigurable logic to increase the performance of applications in comparison with the systems based on standard X86. Due to the programmable nature, the hardware allows the architecture to be redefined to fit in the respective applications. These reconfigurable instruction sets are called personalities. The system provides some personalities, such as single-precision, double precision vector personalities, financial analytics personality and Smith-Waterman personality, which can be used to accelerate certain applications. However, some applications require specialized functionality, which cannot be offered by the existing personalities, thus the Convey-Computers designed a framework to enable the development of Custom Application Engine Personalities, including extension of the instruction sets that allow custom execution.

2.6.2 Coprocessor Architecture

The coprocessor of the platform consists of three main units: The Application Engine Hub (AEH), the Memory Controllers (MCs) and the Application Engines (AEs). Custom commands, which have been developed for the coprocessor, are implemented in Application Engines FPGAs and the AEs are the only FPGAs that are redefined for different personalities.

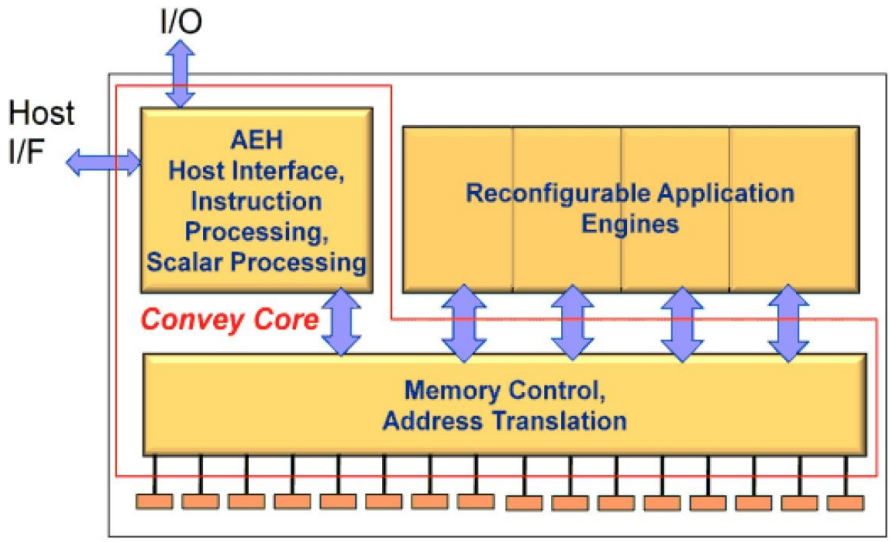


Fig. 2.2 Coprocessor of the Convey platform

The AEs contain four basic interfaces communicating with the rest of the system: the despatch interface, memory controller interface, the managment/ Debug interface and AE-to-AE interface.

2.6.3 Personalities

The user, in order to design a personality, requires the use of PDK that has:

- A set of Makefiles that support the simulation and the synthesis flow design.
- A set of Verilog files for the communication of processor with the FPGAs
- A set of simulation models for all the non-programmable parts of the coprocessor (memory controllers, memory modules)

2.6.4 Memory Controller Interface

The interface Memory Controller (MC) AEs provides a direct access to the coprocessor memory. Each of the 4 AEs is connected to each of the 8 MCs through a DDR interface with a clock frequency of 300MHz. The interface of MC in the AE FPGAs is provided by Convey. Each interface of the MC 8 to AE FPGA is directly connected to a single Memory Controller and all the MC are connected to the 1/8 of the coprocessor memory. The diagram below (Figure 2.3) shows the connectivity between AE-to-MC in the coprocessor.

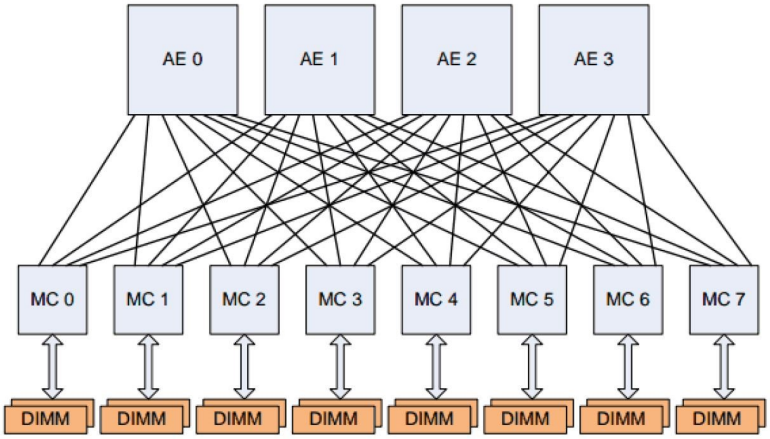


Fig. 2.3 AE-to-MC connectivity on the coprocessor

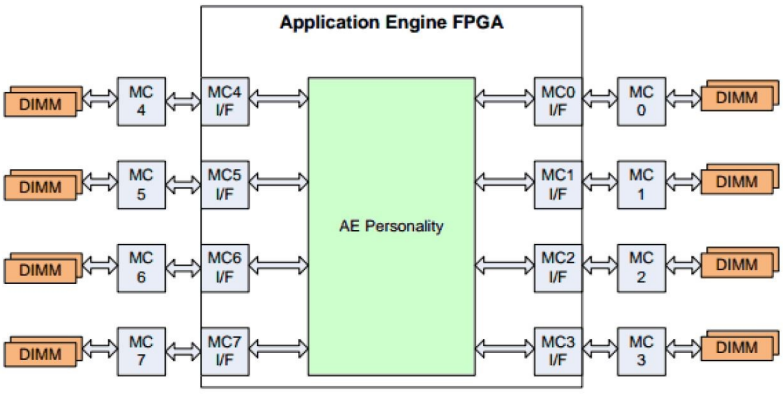


Fig. 2.4 The MC interface connections to the Memory Controllers on the coprocessor

The Figure 2.4 shows the MC interface connections of the AE with the Memory Controllers on the coprocessor. The interfaces of 8 Memory Controller located on the left and right side of the AE FPGA. Each memory controller is connected with 2 DIMM modules. The AE personality is responsible for decoding the virtual memory address, thus only applications for a particular MC are sent.

2.6.5 Memory System

The Convey memory system uses Scatter / Gather DIMMs, which has 1024 memory banks.The banks are spread across 8 Memory Controllers. Each memory controller has two 64-bit buses and each bus has access to eight individual buses (8-bit per sub-bus). Finally, each bus has eight banks. The 1024 banks obtained as follows:

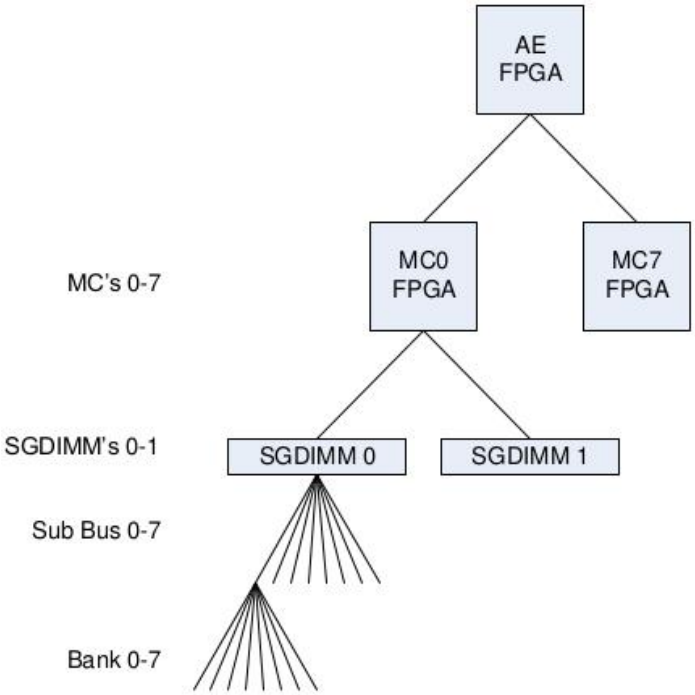


Fig. 2.5 Memory Hierarchy

$$MCs * 2DIMMs/MC * 8subbus/DIMM * 8bank/subbus \tag{2.1}$$

The above Figure 2.5 shows the coprocessor memory

Chapter 3

Related Work

3.1 Introduction

As mentioned in previous chapters, this master thesis elaborates the acceleration of the ScaleJoin [7] algorithm, which is the state-of-the-art within the stream processing family. This group of algorithms has been extremely attractive in various Big Data workloads. This chapter presents the related works on stream join processing algorithms and classifies every work based on the underlying platform. In particular, paragraph 3.2.1 presents the software-based and paragraph 3.2.2 overviews the hardware-based implementations of stream join processing algorithms. Special reference about the ScaleJoin [7] will be provided in the next chapter alongside with every software and hardware-based implementation, which will be used in Chapter X to evaluate the performance of the final system.

3.2 Stream Join Processing Implementations

Due to the growing need for fast high-volume data processing within Bid Data area, the conventional data processing model which involves finite amount of data being stored in the disk and processed by the final system, is replaced with the stream data processing model. Therefore, various methods of acceleration have been used, such as multi-core CPUs, field programmable gate arrays (FPGAs) or massively parallel processor arrays (MPPAs), [19], [6], [22], [20], [15], [9], [16], [17], [12] and [18]. aiming to exploit the parallelism of each method and maximize the processing throughput. One specific and fundamental operation in stream data processing algorithms is the stream join operation. This operation includes multiple streaming data of different inputs being processed according to the processing criteria of each algorithm. This processing model is applied in many applications, such as

sensor networks[citation] in which the streams arriving from different sources may need to be related with any another stream.

3.2.1 Software based implementations

First, we mention some software-based implementations for the stream join problem.

Handshake Join

Handshake Join [22] main goal is to scale out to very high degrees of parallelism in order to avoid the bottleneck that occurs with traditional approaches. Handshake Join operation resembles with soccer players. Similar to the soccer players who are shaking hands before every match and walking in opposite directions, the Handshake Join compares both streams, flowing in opposite directions in parallel. This implementation is scalable enabling the use of multiple cores to increase the window size without limiting performance. Handshake Join replaced the classical three-step procedure of the until then stream join implementations. Figure 3.1 shows how Handshake Join operates.

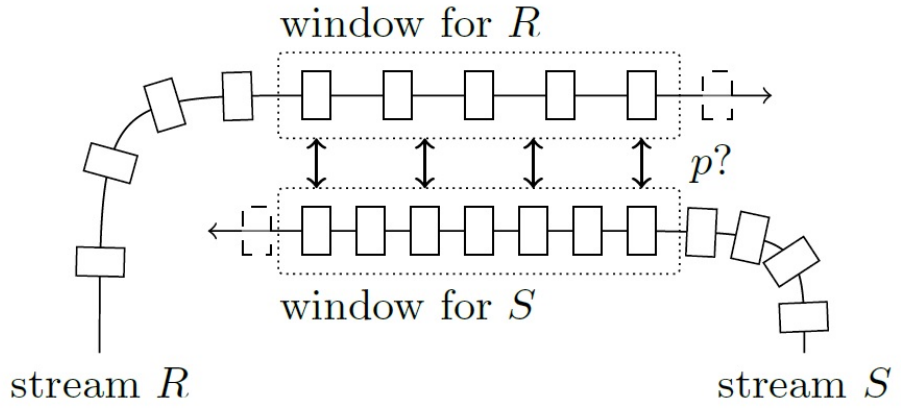


Fig. 3.1 HandShake Join Opearation

Low Latency Handshake Join

Low Latency Handshake Join [20] aims to maintain the positive characteristics of the original algorithm (scalability and high throughput) while reducing latency and producing a deterministically ordered output. The high latency of the original algorithm is caused due to the queuing of tuples along the distributed windows. To avoid that bottleneck Low Latency Handshake Join forwards newly incoming tuples and stores them into node-local window on the core they belong. This way each incoming tuple is "seen" by all involved processing units, it is stored in the respected node-local window and then discarded. After all the comparisons are made with a specific tuple it is removed from the node by an expiry message. Finally a punctuation mechanism with very little overhead, is used to sort the output data and offer a deterministic result.

SplitJoin

One of the latest proposed algorithms for stream join is SplitJoin [15] algorithm. SplitJoin's main advantage is the split of the sequential operation model. More specifically instead of following the classic procedure, to first store and then process the newly incoming tuples, SplitJoin simplifies these computation steps as two concurrent and independent steps ("storage" and "processing"). This splitting allows parallel execution and high scalability which can be further improved by dividing the sliding window into a set of disjoint sub-windows each one assigned to a different join core through a distribution tree. In addition to the splitting of the join computation SplitJoin introduces a single top-down data flow instead of the bi-directional data flow models (Handshake Join). This way there is no linear latency overhead as the join cores are no longer chain connected but completely independent, avoiding as well the inter-core communication overhead.

3.2.2 Hardware accelerated implementations

CellJoin

CellJoin is an algorithm following the three-step procedure for stream join, described in [6]. CellJoin is mapped to the developed by Sony, Toshiba, and IBM alliance, cell processor, which is a heterogeneous multi-core architecture parallelizing the scan task over the available processing units achieving high performance. The idea as shown in Figure 3.2 is to partition the in-memory window of one stream and assign each partition to the available CPU cores so that scan is parallelized. Although CellJoin achieves low latency its scalability is very limited due to the re-partition needed every time a new tuple arrives. The overhead of window

re-partitioning grows linearly with the core number and the input stream rate causing CellJoin algorithm to scale poorly with this kind of architecture.

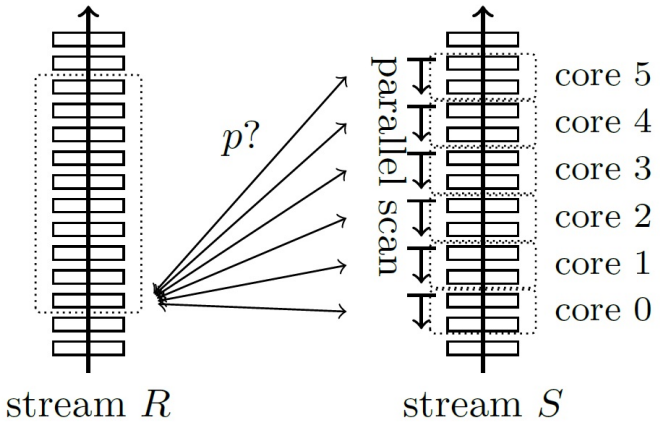


Fig. 3.2 CellJoin Opearation

Handshake Join Mapped on FPGA

The basic idea of Handshake join is to execute window based stream join operation on two streams flowing in opposite directions in order to achieve high scalability and parallelization. To implement this concept on hardware, three issues have to be taken into account. First of all the result collection, secondly the capacity of output channel and thirdly the limitation of internal buffer sizes. The core of the proposed architecture [22] is the join core which is the most fundamental component of the architecture. It's purpose is to evaluate the join condition over the input tuples and generate the output that form the final result. The generated results are then driven into a network of mergers which merge the valid result data streams into one. This network is in fact a binary tree of mergers whose number is determined by the number of join cores. Finally an admission control is used in order to avoid result losses due to buffer overflows. An overview of handshake implementation in reconfigurable logic is shown in Figure 3.3.

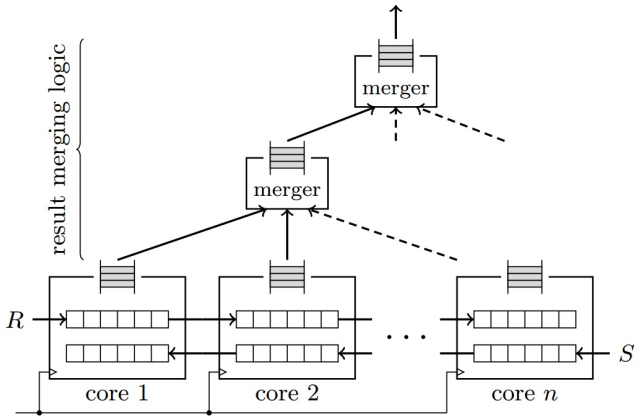


Fig. 3.3 Architecture of FPGA-Based Handshake Join

Improved FPGA Based Handshake Join

An improvement of the Handshake join implementation on FPGA mentioned before is the Handshake Join Operator [17]. Although it is stated in [22] that the merging network supports high degrees of parallelism and that it is high scalable, the truth is that it is an overwhelming bottleneck for scalable performance. Due to this fact an adaptive merging network is proposed which removes the FIFO buffers in the network and introduces a ring structure which is directly connected to the join cores. This way, an output result tuple from a join core is always forwarded and stored in a FIFO buffer (before the merging network), regardless of whether or not this FIFO is the closest path avoiding buffer overflows. The top-level of this implementation of Handshake join is shown in Figure 3.4.

ScaleJoin TUC First Version

This was the first presented FPGA-based architecture that maps the most performance-efficient stream join algorithm, ScaleJoin [7], on reconfigurable logic [12]. The system was fully implemented on a hybrid computer and its experimental performance evaluation showed by up to one order of magnitude speed up in comparison to a fully optimized software-based solution running on a high-end 48-core multiprocessor platform, by exploiting the high parallelization of present-day FPGAs.

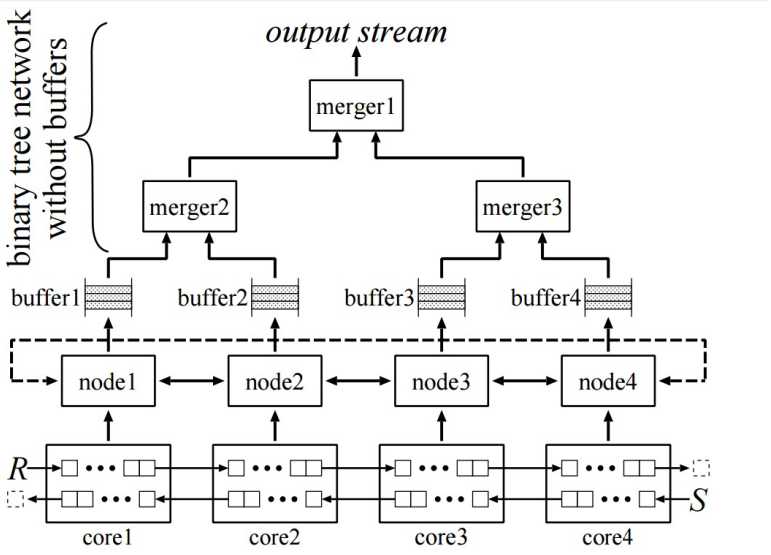


Fig. 3.4 Architecture of Improved FPGA-Based Handshake Join

ScaleJoin TUC Second Version

Based on the prominence results achieved by the first version described above, a second FPGA-based accelerated version [5] has been developed and proposed for the ScaleJoin [7] algorithm. This version extended the first FPGA hardware implementation. Although the first hardware implementation achieved high throughput and scalability it suffered a low resource utilization. The second version proposed a novel architecture and achieved even greater level of parallelism and exploited. It was also mapped on the same hybrid computer. Its experimental performance evaluation showed that this system outperformed the corresponding software-based solution and greatly improved the first hardware implementation.

Both previous solutions are up to now the best FPGA-based solutions outscoring with a high margin any other software based. Therefore, both are great candidates for reference and comparison with the system developed during this thesis. Furthermore both previous systems have been mapped on the same platform , which is the same platform which we are also targeting, facilitating a straight comparison which will result useful conclusions. For this reasons both previous solutions architecture with their strength and weaknesses are more detailed analyzed in following chapter.

Chapter 4

Scale Join Algorithm And Previous Hardware Implementations

4.1 Introduction

In this chapter an in depth analysis of ScaleJoin algorithm [7] is presented, ScaleJoin is one of the most efficient algorithm in data stream join operations. A detail description of how ScaleJoin works, the components of which it consists of and the performance results are some of the main features that will be presented. Also, two hardware implementation of ScaleJoin are going to be presented and analyzed, in order to note the improvements needed. Finally, based on the observations made we propose our architecture which improves on both previous solutions in processing speed and throughput.

4.2 The Algorithm

ScaleJoin is a state-of-the art stream join processing algorithm presented by Gulisano et al. capable of efficiently processing in burst and rate-varying data streams. Its main characteristics are the deterministic and skew-resilient stream processing while also the fine-grain parallelism enabled to achieve high throughput and low latency stream processing. It is build around an abstract data structure, ScaleGate, which operates at the articulation points maintaining the tuples being consumed and produced in a deterministic fashion regardless of the number of processing units or the number of physical streams delivering to them. Furthermore, it allows for the parallel execution of an arbitrary number of sequential stream joins while distributing the overall work among the available processing threads, without assuming any centralized coordinator.

4.3 Algorithm Characteristics and Contribution

The main problems and challenges that ScaleJoin algorithm has to overcome are deterministic processing, disjoint parallelism and skew-resilience. Below is a brief description of each one.

4.3.1 Deterministic Processing

Deterministic is the requirement of given the same inputs in a method, the same outputs must be produced independently of its environment. This requirement allows for a parallel stream join to be leveraged in sensitive scenarios (fraud detection or business-centric pricing applications) and to leverage fault tolerance mechanisms, in which deterministic processing ensures consistency among primary and replica nodes.

4.3.2 Skew Resilience

The ability to process any varying rate of incoming tuples independently by each processing thread. Also the process of a possible different number of physical streams delivering tuples should not affect the functionality of the algorithm but tuples' comparisons should be assigned to a unique processing thread.

4.3.3 Disjoint Parallelism

In of the main algorithm features is that it does not rely on a centralized coordinator to assign the workload on processing units sequentially. Every processing thread available on a thread pool (n total processing threads) executes one comparisons splitting evenly the overall workload. This kind of architectures support $\frac{1}{n}$ degrees of parallelism are leveraged achieving higher throughput and lower processing latency.

4.4 ScaleGate Data Structure

As already mentioned, ScaleJoin introduces ScaleGate, an abstract data type responsible of maintaining the tuples being consumed and produced in a deterministic fashion, regardless of the number of processing units or the number of physical streams delivering to them. Inspired by skip lists, a multi level pointer mechanism is designed and adopted to the ScaleGate requirements. This way, fine-grained synchronization is achieved which in its turn boosts parallelism. ScaleGate is a lock-free implementation ensuring system-wide progress, by guaranteeing at least one of the threads operating on the data structure to make progress

independently of the behavior of other threads. It is also linearizable, meaning that every method call should appear to take effect at some point between its invocation and response. Given these attributes, ScaleGate is able to define a total order in the execution, which is consistent with the real-time ordering of operations, maintaining items in a sorted manner.

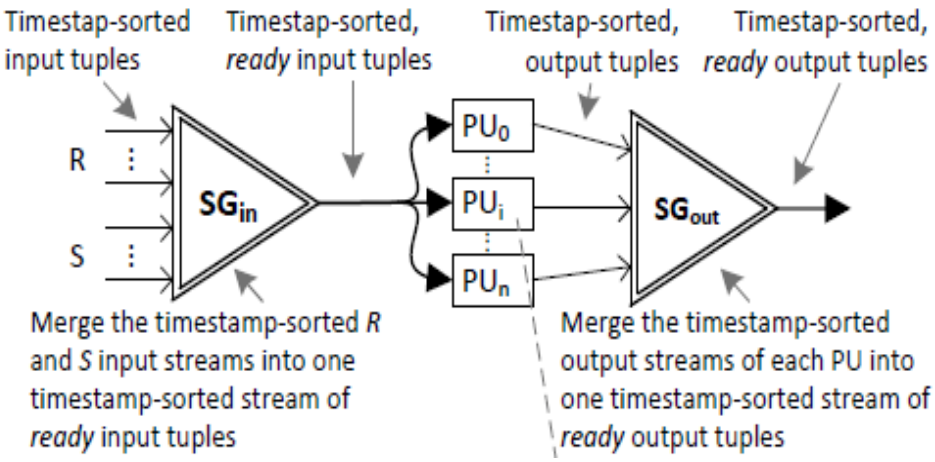


Fig. 4.1 Overview of ScaleJoin's architecture [7]

4.5 Algorithms Functionality Overview

ScaleJoin's main strength is the ability of parallel execution of an arbitrary number n of processing threads (PTs), each consuming, comparing and matching the input tuples delivered by two incoming streams. This process is based on a three-step procedure. In more details, the first step, which is is responsibility of the input ScaleGate structure (SG_{in}), is to merge the input tuples of each stream into a single timestamp-sorted stream of ready R and S tuples as shown in figure 4.1. A ready tuple is defined as ready to be processed, if assuming a tuple t_i^j to be the i-th tuple from a timestamp-sorted stream j, $t_i^j < merge_{ts}$, where $merge_{ts}$ is the minimum among the latest timestamps from each timestamp-sorted stream j. The ready tuples afterwards are sent by the ScaleGate to the PTs in a round robin fashion. The PTs in their turn compare the incoming ready tuple, using a predicate P, with each tuple of the opposite stream inside a window size interval from the tuple. If P holds, an output tuple is created, combining the tuples from the two streams and it is sent to the ScaleGate at the output of the architecture (SG_{out}). SG_{out} then sorts by timestamp the incoming output tuples and sends them as the final system output.

Table 4.1 throughput achieved by ScaleJoin algorithm mapping on tow different target systems.

Window(min)	S1 throughput (t/s)	S2 throughput (t/s)
5	5,100	3,000
10	3,500	2,100
15	3,000	1,750

4.6 ScaleJoin System Evaluation

This section presents ScaleJoin's performance results. It is compared with the up to now state of the art stream join algorithms, CellJoin and Handshake Join. The comparisons are focused on three critical for this kind of algorithms criteria, The first criterion is the system's scalability in terms of comparisons/sec (c/s) and tuples/sec (t/s). The second one is the rate at which tuples can be added to and retrieved from ScaleGate instance from the system and finally the end-to-end processing latency. For the algorithm evaluation two different target platforms used. Their specifications are listed bellow.

- The first platform (named S1), combines 48 cores over 4 sockets of a 2.6 GHz AMD Opteron 6230 equipped with a non-uniform memory access (NUMA) and 64 GB of main memory.
- The second platform (named S2), is based on a 16 cores over 2 sockets at 2.0 GHz Intel Xeon E5-2650 platform with 64 GB of memory.

The data set used is the same used in Handshake join [22] [20] and CellJoin [6] evaluations. Two input streams are used, stream S which its tuple attributes are $\langle$ts,a,b,c,d$\rangle$, where a,b,c,d, are integers, float, double and boolean respectively and stream R which is composed by attributes $\langle$ts, x, y, z$\rangle$, where x,y,z are of types integer, float and char respectively. The predicate used to confirm a valid comparison is shown in equation 4.1.

$$|x-a| < 10 \text{ AND } |y-b| < 10 \tag{4.1}$$

The achieved results for each system are approximately 4 billion c/s for S1 and 1.4 billion c/s with a potential to reach 1.9 billion c/s (with balanced workload) for S2. The throughput regarding the number of tuples per second (t/s) is presented in the table below for different window sizes of 5, 10 and 15 minutes respectively.

When compared with Handshake, ScaleJoin's maximum number of comparisons per second increase linearly when executed in S1. When all the 48 cores are used ScaleJoin

achieve 2.5 billion c/s more than Handshake join. Concerning S2, a step up is noted when the number of PTs exceeds the available cores and hyper threading technology takes place leading to 1 billion c/s more than Handshake join. When it comes to ScaleGate performance, we see that throughput is not limited by the two ScaleGate instances at the articulation points of the architecture. When executed on S2, ScaleGate's rate is increasing with the number of sources and does not degrade when more sources are added. More specifically, for a window size of 15 minutes ScaleGate's rate reaches 150,000 t/s which is 50 times higher of the highest throughput achieved. Finally when it comes to latency ScaleJoin achieves great results. Measurements have shown a maximum latency of 70 ms which is by far better than the original Handshake join which can reach up to 7.5 minutes latency. ScaleJoin's latency increases linearly with the number of PTs. This is due to the fact that the more PTs available the lower the output rate per PT is.

4.7 ScaleJoin FPGA Accelerated Solutions

In the remaining chapter an effort of describing the basic architecture of the two FPGA accelerated solutions ScaleJoin [7] previously proposed [4] [5]. Furthermore, an effort to analyze the strength and weaknesses of both of them and also present some points of improvement which are implemented during the development of this master thesis.

4.7.1 ScaleJoin First FPGA based Proposed Solution

This section presents the first proposed reconfigurable architecture solution for the ScaleJoin algorithm, which was developed by Kritikakis in the context of his diploma thesis in the Microprocessor and Hardware laboratory (MHL) at Technical University of Crete (TUC). It is also published in [12].

The proposed ScaleJoin system consists of two processing elements (PEs) that work in parallel, as shown in Figure 4.2 Each one of them correlates N newly arrived tuples of a single input stream with the all the tuples from the other stream.

This system has broken the processing phase is broken into stages. Firstly, the newly arrived tuples for both streams are loaded to the corresponding processing elements (PEs). Next, the tuples from the S and the R streams, which are not outdated, are streamed to the corresponding PE. The processing units(PUs) inside the PEs compare the two incoming tuples and if the comparison result is "successful", a new merged output tuple is created. The output information is kept into a FIFO at each PU, which is passed through a network of multiplexers to the PE output. When all tuples are streamed and no other results have to be

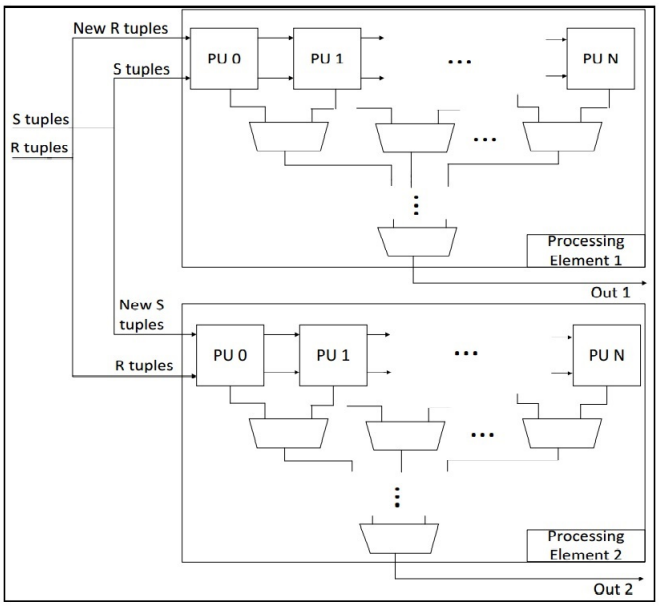


Fig. 4.2 First FPGA solution overview presented [12]

sent out, then the processing phase finishes. In case the newly arrived tuples are more than the available N PUs at each PE, the above process is repeated.

The proposed solution is mapped to 4 FPGAs and each one maps a ScaleJoin module, which has 256 PUs. This system parallelized the problem by loading different newly arrived tuples into each one of the available ScaleJoin modules. Thus, that system could process in parallel 1024 newly arrived tuples. Initially, the newly arrived tuples are stored in shared memory by parallel threads. Next, the tuples are loaded from the RAM and they are streamed to the processing elements via FIFOs. The PUs are connected in a pipelined way, in order to make all the comparisons needed with the minimum amount of memory reads. Finally, each ScaleJoin module outputs the results into an output FIFO and then the results are stored to the global shared memory.

The proposed system is a solution for even higher throughput rates of the incoming streams. Specifically, the high level of parallelism that hardware can offer and the high bandwidth data I/O links that our proposed platform offers, leads to the fact that the reconfigurable part can be reloaded with newly arrived tuples at the same rate-based portion of time, i.e. second. This reloading process can take place many times during the same rate-based time portion.

4.7.2 ScaleJoin Second FPGA based Proposed Solution

This section presents a brief overview of the second proposed solution architecture. This solution was also developed in TUC by Karandinos Ektor [5] during his diploma thesis. It is also mapped on the same target platform and it is a major improvement on the previous work [12] in respect of the processing throughput and the recourses utilization achieved.

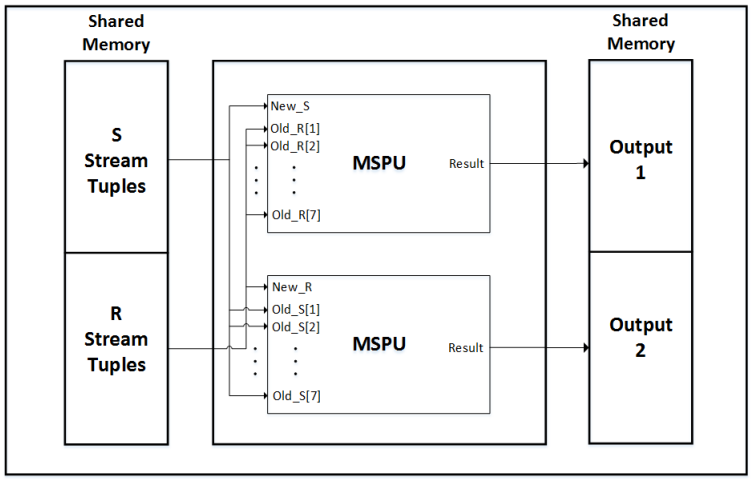


Fig. 4.3 Top Level

Fig. 4.3, presents the the top-level of the proposed implementation which is mapped in each one of the four available FPGAs. It consists of two identical Multiple Stream Processing Units (MSPUs), where each one of them is responsible to make the comparisons between the new tuples of one stream with all the tuples of the opposite stream. Each of the MSPU module consists of seven Stream Processing Units (SPUs), with N Processing Units (PUs) each. The number of SPUs is due to the fact that 8 memory channels are available for each MSPU module. One memory channel is responsible for reading the "new" tuples and storing the results to the shared memory, while the other seven are used for reading and streaming the "old" tuples that belong in the processing time window. More channels for the "old" tuples were assigned, as they comprise the bigger percentage of the computational cost. This way, higher level of parallelism achieved and the process load is distributed to each SPU. When the system starts, the first step is to load the new tuples of one stream to each SPU. This is the responsibility of the first control unit, which also checks if all the new tuples are loaded or if the available PUs in the SPUs have all been occupied. If any of these happens, the execution moves to the next step, which is the streaming of the "old" tuples. At this point, seven control units take over and start feeding each SPU with data ("old" tuples of the opposite stream). Along with these seven control units there is one more running simultaneously, controlling the results coming out. This control unit searches every clock cycle in a round robin way, if

there is a result available in any of the SPUs. The result is then forwarded through a network of multiplexers in order to be stored in the shared memory. It is also important to note that when all "old" tuples have been streamed and all the comparisons have been made, the main control unit is responsible for keeping the system busy as long as it is necessary for the SPUs pipeline to become empty and all results to be stored. After this process is done, the control unit brings the next "new" tuples and the processing continues.

The Stream Processing Unit (SPU) is the next module down to the hierarchy. It consists of N Processing Units (PUs), which are connected in a pipeline way, as shown in Fig. 4.4. The SPU also contains a control unit, which is responsible for the correct function of the module, a network of multiplexers and a FIFO for storing the results. In every clock cycle each PU compares a newly arrived tuple from one stream with a tuple of the opposite stream inside the processing window. As the "older" tuples are streamed through the pipeline, all the necessary comparisons take place and the process ends when all the comparisons have been made.

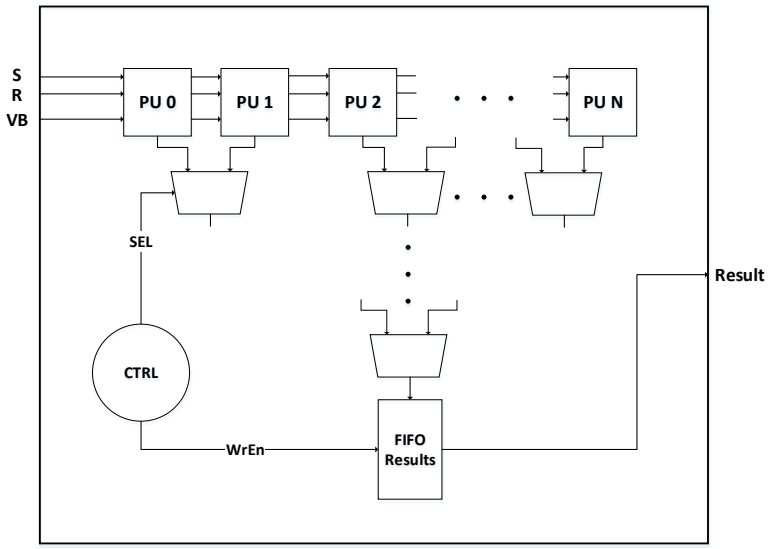


Fig. 4.4 Stream Processing Unit (SPU)

In the innermost level of the system lies the processing unit (PU). This module is fed with tuples from two input streams, i.e. S and R, it compares their attributes and defines if the comparison is successful. The Compare unit is responsible for the correlation-comparison between the tuples' arguments. In order for the proposed solution to achieve high clock frequency the architecture of the PU is pipelined. The execution process is split in two

pipeline stages. The former consists of the registers keeping the tuples and a valid bit as well as the compare unit while the latter includes the FIFO where the results are kept.

4.7.3 Proposed Solutions Drawbacks and Improvement Exploration

The previous architecture and the first FPGA-based implementation of the ScaleJoin algorithm, i.e. ScaleJoin [12], managed to exploit the FPGA enhanced parallelization and the high memory bandwidth available. Both implemented systems outperformed the till now proposed software-based implementations. But,in both solution the resource utilization seems to be really low indicating even that there is still space for performance improvement. Based on this observation and keeping in mind the rate in which new found streaming applications demand more and more processing power, we came up with a new approach, where we managed to increase the number of parallel comparisons that are executed per second.

The basic limiting factor in both previous solutions seemed to be the high routing logic required resulting in this low resources utilization. Furthermore both previously proposed solutions designed strictly around the targeting platform leaving no much space for future extendability. This thesis proposed solution all though is mapped in the same target platform is characterized by great extendability for future mapping on next generation platforms. A feature properly useful now that High Bandwidth Memory (HBM) based systems are available equipped with even larger FPGAs. Finally, our proposal in contrast to the previous ones is fully configurable and programmable allowing for dynamically changing the number of parallel processing units operating on the incoming streams.

Chapter 5

System Design

5.1 Introduction

This chapter analyzes the design and implementation of the proposed system architecture in reconfigurable logic. The HC-2^{ex} hybrid supercomputer was used as the implementation mapping platform. We implemented a hardware design to run on the coprocessor, and a software program controlling the coprocessor and managing I/O. Afterwards, we will present the basic differences between the proposed work and previous solutions, explain the problems and the difficulties that we came across and list some major decisions, which were taken before and during the implementation of the design. In addition, the detailed description of the input/output data is referred. The previous FPGA-based implementations of the ScaleJoin algorithm, i.e. ScaleJoin TUC [7] presented in Related work chapter, managed to outperform the fastest software-based implementations.

The main motivation for the current proposed work rises from the observation that both previous works does not fully utilize the available FPGA recourses neither the high memory bandwidth provided from the target platform. To capitalize on these observations we came up with a new approach, which is scalable and generic and offer impressive performance results, as it takes advantage of the fine-grained and the coarse-grained parallelism that reconfigurable hardware can offer and exploit the high memory bandwidth available. The proposed solution utilizes multiple parallel systolic array structures, which in the most efficiently way perform multiple computations per I/O access. As a result the number of parallel comparisons per second that this system can perform is hugely increased.

The limiting factor in both previous hardware based solutions seemed to be the high routing logic required for the network of multiplexers, which were needed to pass the results from each comparison module to the next level module. This network of multiplexers is

vital because on every clock cycle more than one matches can simultaneously occur. The proposed architecture eliminates this network by making a trade between space and time.

The system has been developed and mapped on all four available FPGAs of the target platform. In the next sections of this chapter a detailed presentation of the architecture follows, most of the analysis is focused on one FPGA because it is the same for all four of them, the same programming bitstream is loaded on each target device.

5.2 Design Challenges

This section is devoted to the challenges met and the major decisions made for overcoming them during the development and the writing of this dissertation. With a really efficient software solution and two well developed hardware solution the speed benchmark for this dissertation was set really high. As already mentioned the target of this dissertation was a system which would be capable of reaching higher processing speed and throughput, more efficient resources utilization compared to the previous works and it would be generic enough for future extension and usage. Taking this requirements into consideration, the main challenges met during the system's design and implementation are described in detail through the following paragraphs.

5.2.1 Increase the Number of the Pipeline Array PUs

The nature of stream processing problems and stream join especially matches perfectly to systolic array architectures. This is due to the vast number of operations needed to be done on a strictly timing window, and that is exactly the systolic arrays' strength. As a matter of fact, both previous hardware based works proved that by the speedup managed by both proposed solutions and especially, the second work which achieved higher speedup performance, by increasing the number of overall processing units. FPGAs are suitable for mapping systolic arrays of large width in their resources, combining multiple processing units, which are capable of working in parallel. Both previous solutions showed poor resources utilization, meaning that the available platform had still more performance benefits for extraction.

In the context of this work that was the main challenge we had to overcome. After studding and analyzing deeply the previous works we concluded that the main reason of the aforementioned poor resources utilization was the combination of the large network of multiplexers and the FIFOs on each processing element. This network is shown in the red circle in Figure 5.1. This network exists in order to cover the case when more than one processing units produce a result and this result has to reach the systems' output.

This particular network consumes a large amount of the available routing resources and makes the design placement on the FPGA really difficult and in some cases, when the number of PUs exceeds a certain number, impossible. When this threshold is reached, even after of hours of effort by the tools to achieve a placement these designs never produce a final bitstream.

The solution for this problem we come up, was to eliminate this network of multiplexers and free as much routing resources as possible. This aim lead us to the proposed architecture, which is based on the idea of recycling data which have multiple produced results. One more difference of this solution in comparison to the previous proposed, is the completely different memory accessing scheme used in order to achieve better utilization of the available bandwidth, allowing for even higher FPGA parallelization. The proposed solution decouples the available memory access ports of the top level system and follows one more general architecture where each port is attached to each own module which integrates the whole functionality determined by the problem. Of course, this choice requires that each port will serve as both a reading and a writing port. This choice resulted on a boost on the mapped processing units but also it is the core idea behind the generality, configurability and programmability of the final system.

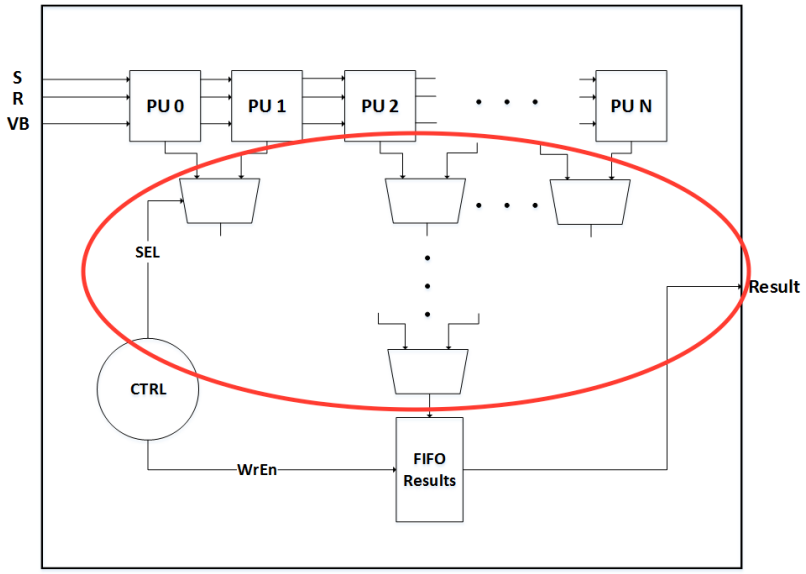


Fig. 5.1 Second solution multiplexer network

5.2.2 Overcoming Routing Problems and Match Timing

The next challenge we came up during the system's development, which was the most demanding on development time, was the need to meet system's timing requirements.

Timing refers to the logic delays between sequential elements. When we say a design does not "meet timing," we mean that the delay of the critical path, that is, the largest delay between flip-flops is greater than the target clock period. Because of the architecture's pipeline nature timing it was really important for the correct system's functionality .

Concerning the two previous works the same challenge was holding. As their timing results indicate the first proposed solution didn't manage to reach the timing goals in contrast to the second one. For both previous works the main cause behind the timing problems they had was again the routing resources competing explained extensively in previous section.

Because the processing system relies on data arriving on every clock cycle from main memory, the main timing requirement is set by the targeted platform's memory sub-system. This timing is fixed to 150 Mhz and there are no margin for discrepancy of it in order of assuring than there will not appear metastability errors.

For overcoming the timing challenge, specific selections about the options of the tools used during the development process, have been made for guiding them in achieving the best possible results. For every tool in the design process the execution options were selected with maximizing timing in mind. Also, extra timing closures are added in the constrains group of the system for leading the tools in the best results. Although the tools are given the most efficient parameters still the timing goal was unreachable. That's why special design treatment was used for on the Register Transfer Level(RTL) code. More precisely the following methods were used in order to reach our target.

- Most of the effort was put in eliminating as much as possible global routing signals like resets and enables. Mostly we focused on rebuffering and distributing these signals through local routing resources. Furthermore, extra effort was placed on balancing the control signal sets of the sequential elements so that the tools will be able to place and map as many of them in the FPGA's configurable logic blocks CLBs.

- Another design choice was to try and map logic on resources that on a different way would have been left unused. For example a respective amount of DSP resources was used as arithmetic units and large counters. Also the usage of these DSPs released a great amount of registers as in this case extra dedicated and integrated in DSPs ones were used.

- Moreover, the critical paths were heavily pipelined by adding register layers to divide combinatorial logic structures.

- Parallel structures for separating sequentially executed operations into parallel operations were used.

- Flattening logic structures specific to priority encoded signals. Priority encoded signals is one of the main reasons of large combinatorial logic.
- A carefully register balancing scheme was chosen to redistribute combinatorial logic around pipelined registers.
- Hugely reordering paths effort was conducted to divert operations in a critical path to a noncritical path.

5.2.3 Efficient Operation Scheduling For System Maximum Utilization

Although transforming the system's architecture from area centric, like the previous proposed solutions to a timing centric one gave the chance for better routing of the design it also raised a critical challenge. For the proposed system to be working on its maximum capabilities a lot of effort was placed on getting the best possible resources utilization. Because this system is based on the idea of computation repetition for some data, which are stored temporary on memory structures, dedicated control logic was developed on every level of system hierarchy targeting as much efficient resources sharing as possible. For example the memory port used by each processing module is carefully arbitrated among its requestors and dynamically changes direction (ether loading or storing data) so than no system clock cycles lost. Also the memory port sharing is done with having in mind that the memory sub-system performs its best when used on burst mode. The same scheduling principles also stand for the main processing unit (the systolic array), which needs to be shared among the data coming for processing from the main memory and the data waiting for re-processing in the temporal storage structures.

5.2.4 Generated Results Parallel Storage Data Overflow and Overwriting Prevention

Considering the system's requirement for a unified produced result data structure. One major issue came up prospecting for solution. The issue raised was that each processing module is completely independent from the others a dedicated memory pool for it's result would be necessary. But it was impossible to know from the application's execution start the exact number of the produced results, so that the appropriate size of memory would be allocated. Furthermore, the parallel and independent processing modules had to share the available common memory space and the danger of overwriting result data was present. The solution

to that was to divide this space in 16 (like the number of processing module) parts and each processing module to take its own part and store its results on that. Although this idea lead to a solution, it generated two critical questions. The first one was how each processing module will know where its memory part starts to store data there and the second one was that because processing module does not have information in advanced for the number of the possible results produced, data overwriting prevent mechanism is needed.

To overcome the first one, special logic has been developed and an internal identifier is given to each processing module. This identifier in conjunction with an input parameter passed to the system by the software API denoting the maximum number of result memory is available for each processing module, makes the calculation of the memory address starting point feasible. This calculation takes place during the configuration step. After the initial address has been generated each result is stored in the first memory position available after the first one. The first memory spot is left empty until the processing finishes, it is used for keeping the number of results produced from the processing module. When all processing finished the processing module stores in the first memory position the exact number of results produced by this particular module. As a result the responsible software procedure for collecting and gathering the results on one common memory space will be informed for the results number.

The solution to the second question is given by using a counter which keeps track of the number of results stored in main memory until that moment. As mentioned before the maximum number of possible results that space is existing in memory for each processing module is known in advanced. That's why an up counter is selected witch notify the control unit to stop generating more requests in case it reaches the threshold set by the initialization parameter, thus each processing module stores results only in each limits and overwriting is prevented.

5.3 FPGA architecture Overview

Figure 5.2 shows an abstract view of the proposed reconfigurable architecture. The system consists of sixteen fully configurable and independent parallel Stream Processor Engines (SPEs). Each of the SPEs operates on a Multiple Instruction, Multiple Data (MIMD) way achieving high processing parallelism. Each module operates asynchronously and independently. The number of SPEs is bounded by the number of available main memory ports. For the whole system 64 independent SPEs were implemented. By assigning on each SPE one memory port we managed to reach the peak of the memory bandwidth.

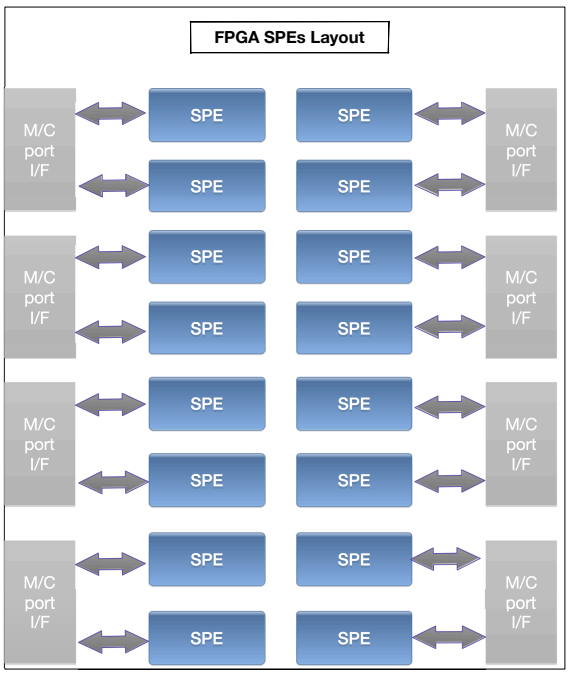


Fig. 5.2 FPGA system layout of the 16 independent SPEs

5.3.1 SPE architecture

Each SPE, as shown in Figure 5.3, consists of six main parts. The Stream Processing Unit (SPU), is the main processing unit, while the configuration registers are used for programming the whole unit. Also there are two memory management units, responsible for loading data from the main memory to the SPU and storing results back to memory sharing one main memory channel. The control unit constitutes the engine brain. Before data processing starts for each SPE an initialization is necessary during this step the configuration registers are loaded. A detail description of the configuration step is analyzed in section 5.4.

The respective engine operates on a repeatable manner, as shown in Figure 5.5. Firstly, it loads the new tuples of the first stream in batches of equal size to the depth of the SPU pipeline, after that all the available tuples of the second stream (Old stream) are streamed through the pipeline. A representative flow chart, showing how the subsystems interact during the configuration step and during one of the execution iterations of the aforementioned loop is shown in Figure 5.4. At this point, it is important to note that the tuples of the "old" stream are much more than the "new" stream tuples. Responsible for the orchestration of the aforementioned operations is the control unit shown in Figure 5.3. Every time one iteration is completed this unit starts a new iteration after it has ensured that the previous operations are completed normally and the produced results are stored to main memory.

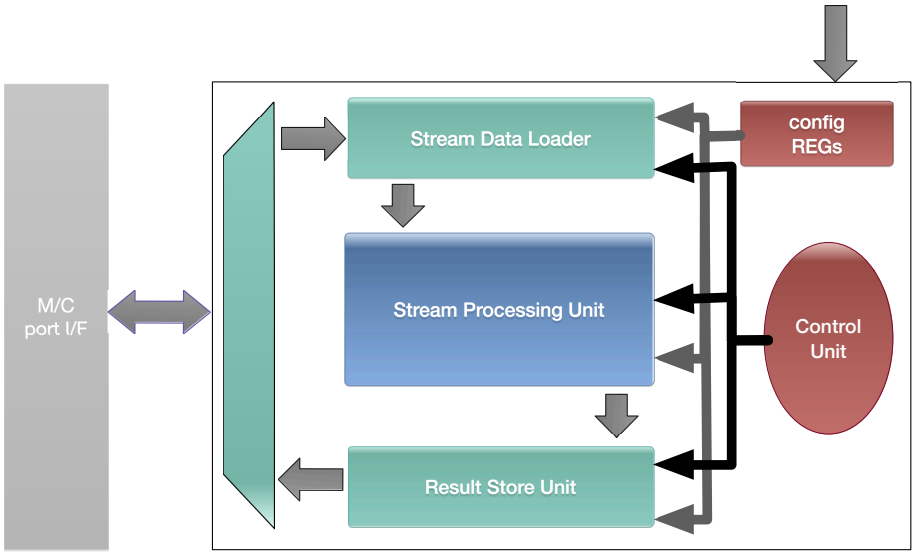


Fig. 5.3 Stream Processor Engine

Finally, a very important unit inside the SPE is the arbiter, which makes possible the common memory port sharing. Its design is based on two major decisions. It arbitrates on a tightly timing manner so that no system clock cycles get lost. Furthermore, in its operation it has to predict possible memory controller stalls and prevent the system of getting in such states, avoiding losing valuable time waiting for store or loading operations to be completed.

The following sections present a detailed architecture analysis of each internal unit separately, starting from the loading unit, which brings the data to the main processing unit (SPU) and finishing to the unit responsible for storing the results to the main memory. The analysis flow through the next sections follows the system's data flow.

5.3.2 Data Fetch and Dispatch Unit Architecture

The first unit on the data stream flow is the data fetch and dispatch unit. Figure 5.6 shows an overview of its individual subsystems and their connections. Furthermore, there are plenty critical arithmetic submodules used mostly during the units initialization step. An in depth analysis of the units functionality and its steps is presented in the following paragraph.

The top level unit functions on two steps, the first one is the initialization and configuration, which takes place from the top level SPE through the configuration registers. During this step dedicated arithmetic hardware blocks calculates the starting end ending memory addresses, the batch sizes for both input streams and the initial values for the two ID generation engines. Next, it finally triggers the execution step. In this particular step the control unit, responsible for fetching the raw data, generates and pushes memory read addresses

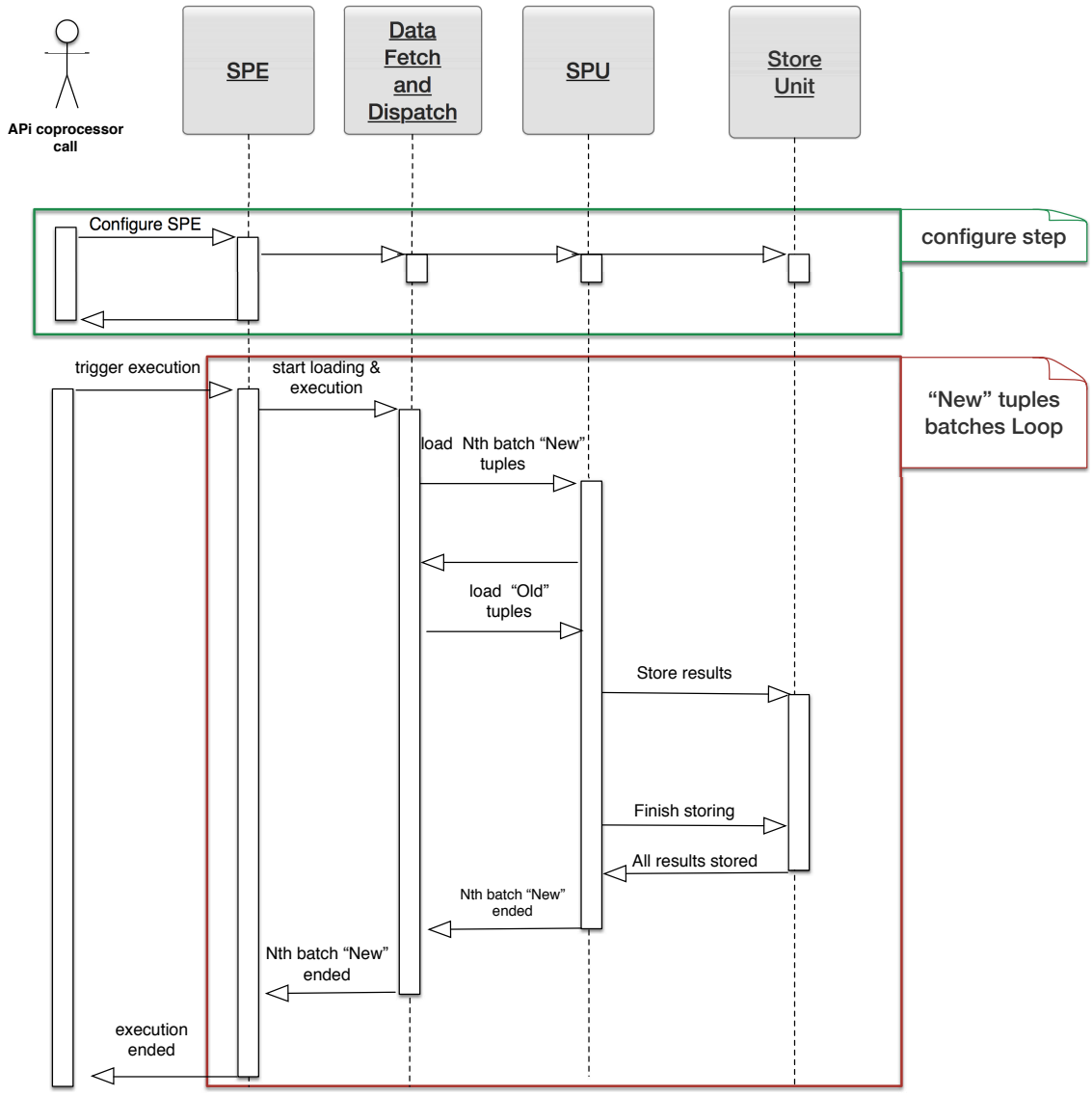


Fig. 5.4 System flow chart

and requests to the memory controller on a burst mode. The control unit, firstly, burst 128 addresses for the so called "new" tuples and at the end it consequently requests for all tuples from "old" stream. The above mentioned process is repeated until all "new" tuples are loaded. At the moment the first data arrive to the FIFO buffer the second control unit (Data Dispatch Control Unit) in combination with the ID generation engines start pushing data and their corresponding meta data, like the tuple id and which stream the data belongs, to the SPU system. Fig. 5.6 shows each stream have a dedicated ID generator. Each generator is mapped to FPGA's dedicated digital signal processing (DSP) module. This module is suitable for the arithmetic operations nature required, as it contains both a multiplier and an adder.

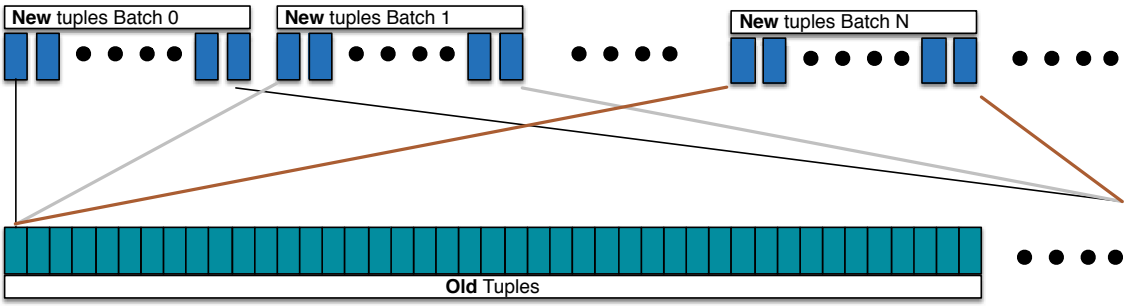


Fig. 5.5 SPE batch mode executing

5.3.3 SPU Architecture

As mentioned above, the main processing element of each SPE is the SPU. This submodule consumes the generated data from the Data Fetch and Dispatch Unit described in the previous section and produces the results for the storing Unit. Each result is IDs concatenation of the "new" and "old" tuple matching the algorithm criterion. The SPU shares the same two steps circular functionality of the previous module. The heart of each SPU is a systolic array structure of 128 Processing Units (PUs). Specifically, it is a computing network processing the following features:

- Synchrony, meaning that the data is rhythmically computed (Timed by a global clock) and passed through the network.
- Modularity, which is meaning that the array consists of modular processing units.
- Regularity, which is the arrays property of homogeneously interconnection of the modular processing units.
- Spatial Locality, meaning that the arrays cells has a local communication interconnection among them.
- Temporal Locality, every cell transmits the signals from one to other on at least one unit time delay.
- Ability for deep pipelining, meaning that the array can achieve a high processing speed.

This systolic array in combination with the required storing elements and logic for rescheduling data processing on later timing stage as well. Each of this element and their corresponding functionality is the topic in the next subsections.

As already described, during the first step "old" stream tuples with their corresponding ids, which identify each tuple's position in the stream, enter the SPU in batches of 128 tuples

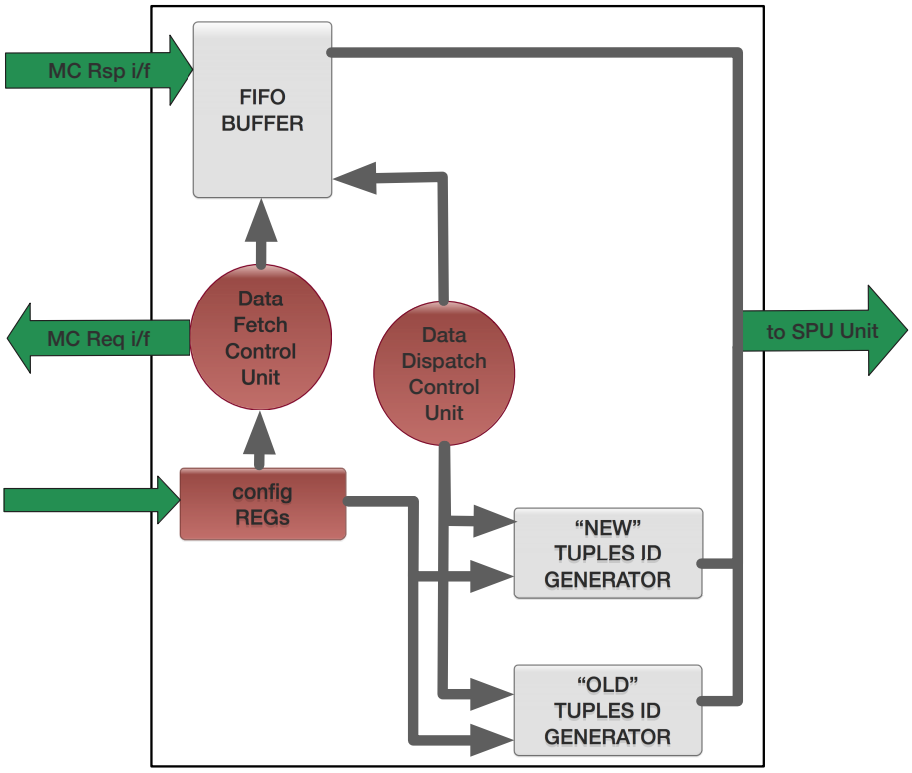


Fig. 5.6 Data Fetch and Dispatch Unit Architecture Overview

matching the length of the array of PUs. The tuple's data are streamed and stored on every array PU, and the ID data are stored in dual port block RAM as shown in figure 5.7. For example, the first incoming "new" stream data is stored in PU's zero reservation station and the incoming ID is stored in position zero of NewID DP Memory shown in Figure 5.7 and so on until all 128 PUs are loaded or the "new" stream has no more data.

The second step is the main processing step, it follows the data loading to the PUs and begins with the "old" stream tuples tagged along with it's corresponding IDs are streamed through the whole array till the last PU. Each pair of "old" stream data and ID enter on two different subsystems, the tuple data travels through the PU array and the ID goes through a FIFO module, named "OldID FIFO" on Figure 5.7, of equivalent length, so that both of them will reach the last position end exit the two subsystem at the same clock cycle. The path followed by the incoming data during this step is shown in Figure 5.7 with green arrow lines.

Every PU compares each own already preloaded during the previous step one "new" tuple with the "old" streamed one. The PU internally keeps track of the times that this particular tuple has entered the array and also the times that the comparison was successful. On each clock cycle these statistics are updated and streamed to the next PU. At the other end of the array, special logic is used to detect if successful comparisons exist.

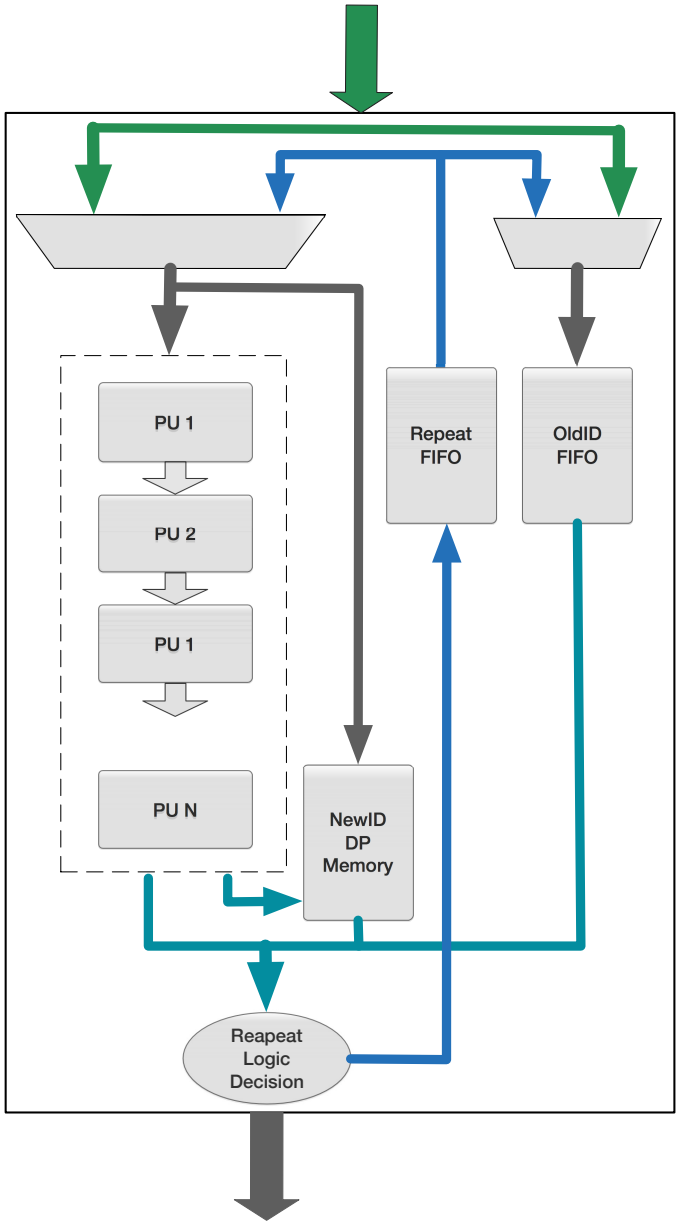


Fig. 5.7 Stream Processor Engine

For every successful comparison inside the array necessary information about the matching PU number is tagged along and is used as a reading address for the “NewID DP Memory” so that the “new” tuple’s ID can be retrieved from this memory. A new merged result output tuple is created and forwarded to the memory storing unit. This result tuple is a concatenation of the two IDs exiting the aforementioned memory and the one at the “OldID FIFO” output.

A hypothetical example of the described functionality is given below.

Considering that a “new” tuple data is loaded in PU four and its corresponding stream position ID is in position four of the “NewID Memory” and it has the value of ten, meaning that it is the tenth stream element. When a “old” stream tuple data enters the array its ID, which has value twelve, is entering the “OldID FIFO” and when the tuple reaches PU number four where it matches with the “new” one, the ID is also in the same FIFO position. The matched PU number is tagged along the tuple through the whole array and when it finally exits the array the carried over number (in this case it is number four) is used as an address to the memory in order to retrieve the stored ID (the stored value is 10). The combination of this particular ID and the ID coming from the FIFO, which has the value of twelve, forms the final result to be stored. The combined tuple is of the form $\langle$ 10,12 $\rangle$

Of course, there are cases when one “‘old” stream tuple can match more than one PU loaded “new” stream tuples. In such a case the bus interface, which on each clock cycle streams the meta data for the matching PU number would be of a variable size unknown from the start. Taking into account that the SPU array is of depth 128, for one hit 7 bits for the transfer bus are enough. On the other hand if there are more hits that bus should be a multiply by 7 wide bus. Because of the nature of FPGAs and hardware in general such a bus is not possible.

The proposed solution for this problem is to transform the problem from the physical resources domain to the time domain by using memory to keep data needing processing for later stages of execution. In this particular case extra logic to the PU modules in combination with a FIFO buffer are used in for achieving this. The extra logic on each PU keeps track of the number of matches and the number this particular “old” tuple has entered the array. When the matching times number is equal to the array entered times number plus one only then the matching PU id is kept and is used as the address of the “NewID DP Memory”. At the arrays exit end there is logic which compares the two numbers mentioned above and in case that they are not equal this “old” tuple data in combination with the ID exiting the “OldID FIFO” and the number they have entered the pipeline are pushed in the “Repeat FIFO” shown in figure 5.7. When this FIFO is full the SPU stales “old” tuples data coming from the main memory and starts feeding the array with data from this FIFO. The dataflow in this case is shown in the figure with blue color.

Keeping up with the previous example, lets consider that the incoming “old” tuple matches except from PU four also to PU number six, when it enters the array for the first time the number of entering times is gonna be one and the matches number zero too. At the exit of the array when it has been passed through all PUs these new numbers will have the value zero for the numbers entering the array and two for the numbers match, the result tuple for the match corresponding PU four will be formed the way previously described but because

of inequality of these two stats an extra tuple will be formed and pushed to the redo FIFO. This new will combine necessary meta data information for the tuple for later re entrance to the array. At some later stage, and more specifically when the redo FIFO is almost full this tuple will re enter the array but in this case the number of entering times is gonna be two and the matches number zero once again. The moment it will reach PU number four the number of matches will become one which is less than the entrance number, so this match will be discarded and only the match in PU six will be consider a valid match, because it is the matching number equal to the entering number. Finally, this equality is also enough for not re entering this particular tuple the pipeline again.

PU architecture

At the beginning of this section the SPU's array of PUs was characterized as the heart of the system, but the cell of this heart is the PU itself. An overview of the PU's module architecture is depicted in Figure 5.8.

For each SPU and the system as a whole, the number of integrated PUs is the basic ingredient for leveling up the tuples processing throughput. Each identical PU is chained with its neighborings, forming a homogeneous network of tightly coupled data processing units, which operate on a systolic array manner. On this system the number of PUs are connected in a linear array topology. For every PU on each clock cycle, new data inputs for processing are coming from the previous one and its outputs make up the required inputs for the next one. Architecturally, the majority of the building blocks of each PU are memory blocks necessary for keeping data and processing results. Furthermore, as figure 5.8 shows, multiple parallel instances of arithmetic units are used, in order to reach the systems strict timing closures. In the previous section a brief description of the PU's functionality has been described, however this section give a more detailed one.

During the first processing step, when the batch of "new" stream tuples are loaded in the PUs, the tuple data are saved on the PU's respective registers and the "Enable" flip flop, visible in the decremented figure is set to one, when this particular PU participates in the second processing step (this also explains the existence of the end gate in the figure). If the enable bit is not set, this particular PU does not produce results, but it only serves a pass through way so that the previously generated results and data are able to reach the end of the PUs systolic array. Coming to the more interesting step while the "old" stream data are streamed through the whole array, for every enabled PU, the tuple data are stored in the "Tuple Old" register and also are participating to the comparison between itself and the already preloaded (from the previous step) "new" tuple data. This comparison takes place in the "Tuple Comparator" module. If there is a successful match from this comparison,

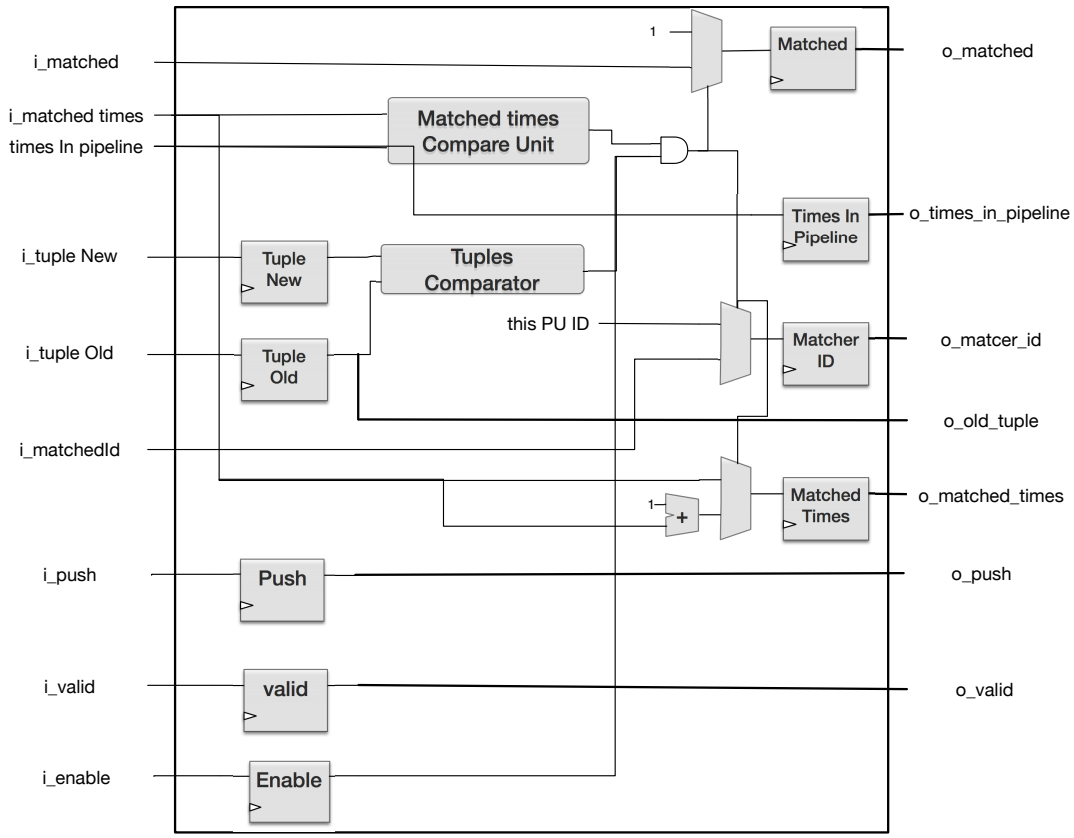


Fig. 5.8 PU internal Architecture

one more condition must be true for the certain match to be considered as the array output result. As described in the previous section and in the corresponding extended example, for a match to be reported at the array output, the incoming data from port “i_times_in_pipeline” indicating the number of this “old” tuple entering the array pipeline and the incremented by one number of matches (incremented because of the current match) should be equal.The number of incremented matches is a result of increasing by one the “i_matched_times” entering data, which hold the matches scored through the pipeline until the data reaches this PU.

When these two conditions are met the “Matched” bit is set, the “MatcherID” register is loaded with the internal ID of this particular PU and the “Matched times” register stores the incremented value. Of course, if the previously mentioned requirements are not met, these registers keep their previous values or are streaming the corresponding inputs when the “Push” bit is set. This bit is the trigger signal keeping the whole array streaming data and operating on them.

At this point it is worth mentioning that two different “Tuples Comparator” modules have been developed, placing different design challenges as its complications and placement have a huge effect in system timing. The first experimental version is an equality comparator, which compares the two input stream tuples for an exact match. Although its hardware requirements was only a comparator, special design treatment was required because of the nature of FPGA internal logic placement in combination with the large width of the two operators participating in the comparison(64 bits each one). Section 5.2 presents an in depth analysis of the challenges that came up during the design process. The second version, is a combination of two identical circuit instances which calculates the absolute difference value of two 32 bit operands and then compares this generated value with a fixed number. In case that this difference is smaller than this predefined value the comparison is considered successful. To summarize for the second version, four 32 bit values enter the describing module and after they are compared by the two internal modules on pairs with the method previously analyzed, if both pairs results to a match only then the top “Tuples Comparator” module also produces a match.

5.3.4 Store Unit

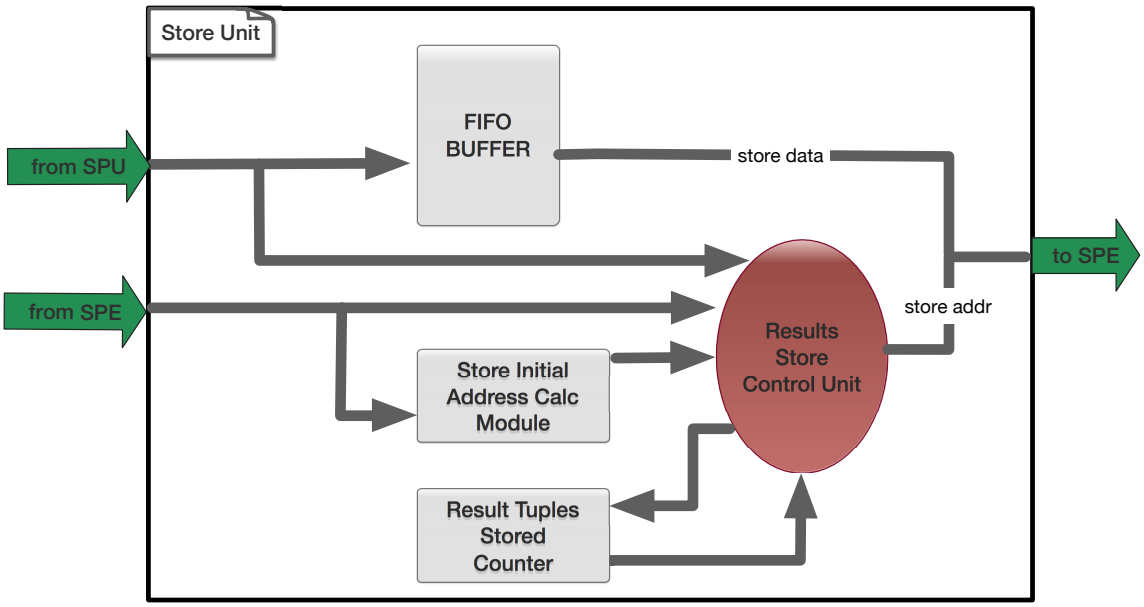


Fig. 5.9 Store Unit Architecture

The final module in the processing chain is the Store Unit, although it is not as complicated as the previous systems it is also considered vital and it has some interesting solutions. Figure

5.9 gives an overview of the systems architecture. The “FIFO BUFFER” is the main component buffering the generated results from the SPU. Its usage makes the memory port sharing between this module and the one loading main memory data possible. Further more, its existence prevents any results lose in case the reference platforms memory subsystem raises any data stall requests. Of course, this FIFO supplies only the result data, the extra store control signal required for a store request to be completed, are coming from the control unit pictured in figure. These signals are the store address and the store request bit.

The other two modules visible in Figure 5.9 are used in order to prevent result main memory overflow and overwriting as analyzed in subsection 5.2.4 of the design challenges section.

The “Result Tuple Stored Counter” is a large up counter initialized every time the execution start to zero and incremented every time a new store request is done until the maximum number of main memory positions available for each SPE. In case of a counter overflow a signal triggers the end of execution of this particular SPE and the control unit sends a final store request , which is the number of results stored till that moment (in this case the maximum possible). When the SPE finishes execution normally, without the number of produced results exceeding the threshold, it signals the “Store Unit” that processing is finished and the control unit again this time stores the number of results have been stored (the counters exit) to the first memory position as before. This step is completed of course after the control unit has insured that that the pending requests within the “FIFO BUFFER” have all been stored.

5.4 SPE Configuration

As mentioned in previous section before each SPE unit has to be configured first in order to start fetching data from incoming streams and producing results. The configuration process is done by assigning proper values to the internal configuration registers.

Each SPE unit have many configuration capabilities, it is possible to be configured as an independent processing engine or as a cascaded extension of its previous or following SPE unit, respectively. Figure 5.10 illustrates four possible configurations of both independent and cascaded cases.

For each SPE the minimum number of “new” stream tuples loaded is equal to the smaller size of the systolic array pipeline, which is the same as the number of PU units connected to form this array. In our case this minimum value is 128 PUs. The final system consists of a total of 64 SPEs. The minimum configuration of our system is 64 SPEs of 128 PUs, meaning that the 64 SPEs will be loaded with the same 128 “new” stream tuples and each one will

load “old” stream tuples with a stride of 64 until every tuple is consumed. This configuration is shown in Figure 5.10a, this figure shows how the SPEs are loaded with the same “new” stream data (shown in white color) and how each SPE independently loads all a portion of the “old” stream (shown in different colors). In this particular case each SPE loads the $\frac{1}{64}$ of the whole window stream. Every time when each SPE finishes loading and processing the current tuples, it continues its operation by loading the next batch of 128 “new” stream tuples and repeats the same process until all data from both streams are related with each other.

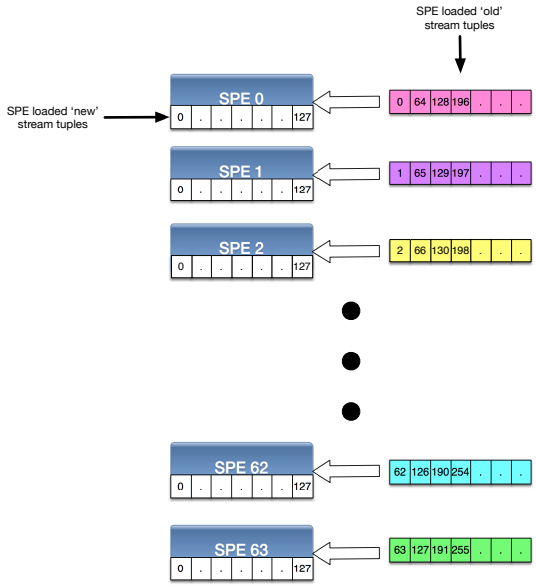
Figures 5.10b 5.10c and 5.10d pictures how multiple SPE can be configured together in order to generate longer arrays. As a result, the number of SPEs operating in parallel on different “old” stream tuples is decreased, thereby the parallelization moves from the “old” tuples stream to the “new” tuples stream. For example, Figure 5.10b visualizes how SPE 0 loads “new” stream tuples from 0 to 127 and SPE 1 loads tuples from 128 to 255 (shown in orange color) however these two SPEs now work as an extension and they load the same fraction of “old” stream data (shown in pink color). The same principles apply also for the configurations in figures 5.10c and 5.10d.

Furthermore, the stream join operation requires both incoming streams to be related with each other. For example, when two input streams S1 and S2 have to be processed, the half of the available SPEs will operate on the S1’s “new” tuples alongside with the S2’s “old” tuples, and the other half SPEs will operate on the other way around.

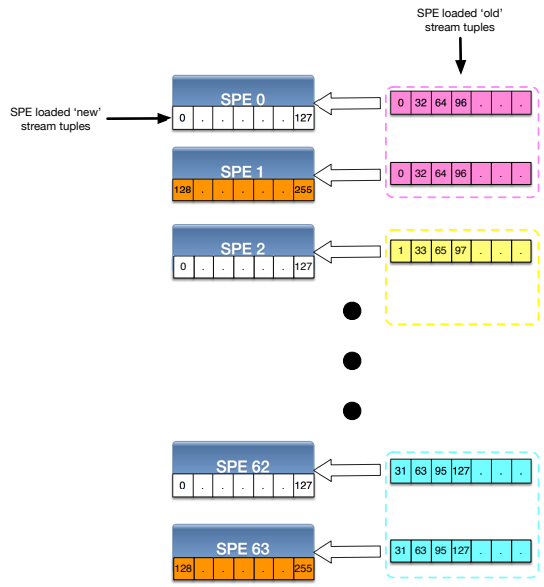
In order to make the final system as flexible and scalable as possible one more configuration option was added on the SPEs. The SPE takes as input the addresses of both input streams and decides on the fly which one is the “new” and which is the “old”. This is achieved by setting the appropriate bit value in the configuration registers. This feature allows the user to decide how many and which exactly SPEs to dedicate on processing on each stream , for the particular system each stream can have from 0 to 64 SPEs working on it.

For configuring each SPE a 16 bits wide configuration register is used, each SPE has its own one. During the configuration phase these registers are configured with the values given by the user through the software API presented below. Table 5.1 shows the fields in the configuration register.

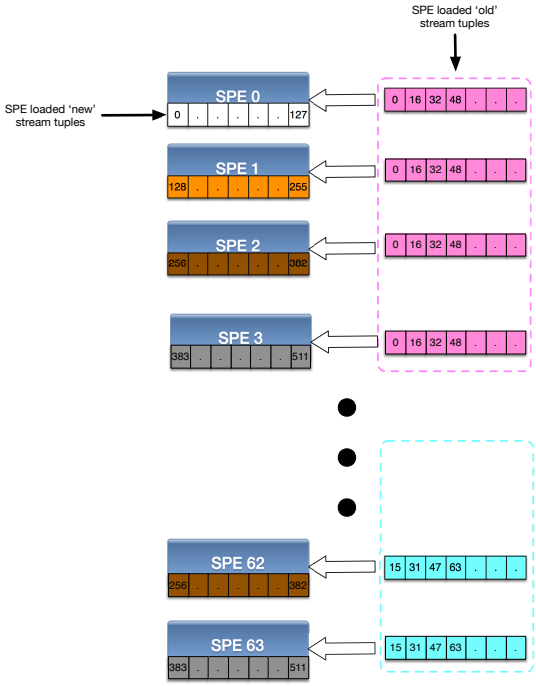
For completing the configuration process one more register is used. It is a 32 bit register which programs the SPE about general configuration parameters needed for generating the suitable address generation scheme for loading the incoming data. A detailed description of this binary fields is shown in Table 5.2.



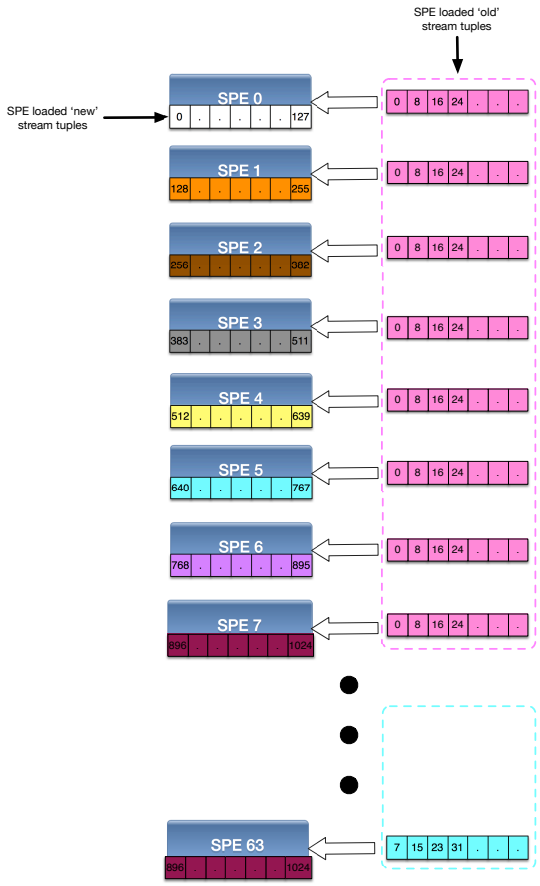
(a) 64 SPEs X 128 PUs Configuration



(b) 32 SPEs X 256 PUs Configuration



(c) 16 SPEs X 512 PUs Configuration



(d) 8 SPEs X 1024 PUs Configuration

Fig. 5.10 Four possible System Configurations

Field Name	Bit Range	Description
EN	15	This bit enables the SPE in case the bit is not set this particular SPE never starts
PID	13:8	The Parallel ID specifies in which “old” tuples parallel group this SPE belongs
STRSEL	7	This bit selects which incoming stream to consider “new”
MODCONF	6	This bit configures the SPE’s mode when it is 0 the SPE is an extension mode
SID	5:0	The Serial ID specifies the position of this SPE in the serial chain

Table 5.1 SPE’s Configuration Register Fields

Field Name	Bit Range	Description
S2SER	31:24	The number of SPEs configure as an extension on stream S2
S1SER	23:16	The number of SPEs configure as an extension on stream S1
S2PAR	15:8	The number of SPEs groups configured to operate in parallel on stream S2
S1PAR	7:0	The number of SPEs groups configured to operate in parallel on stream S1

Table 5.2 System’s parallel and serial modules Configuration Register Fields

5.5 Software Application Programming Interface(API)

The whole system is a combination of a software executed in the commodity CPU available on the target platform and the reconfigurable system described above. The aforementioned system allows the software to offload its mostly computational demanding part to it and take advantage of the high throughput processing capabilities it provides.

The host software running on the CPU creates the data structures required by the application as it reserves the necessary memory for them, initializes them, configures the reconfigurable logic and finally dispatch a request to it for starting processing the data.

In more details the data structures utilized by the application are two arrays holding the newly arrived tuples of both incoming streams for this particular second and all the tuples data belonging to the processing window and an array to hold the derived results. For each of the above structures two memory pools relying on different physical memory are reserved. One is reserved for the host memory and the second one is for the coprocessor's one. This particular choice is made because of the target platform memory system architecture and the need to maximize it's capabilities.

The two pools of memory the platform is equipped with appear to the application as a common linear cache coherent address space. All memory is accessible from both the host processor and the coprocessor, but performance is significantly impacted by the location of the data accessed.

The host processor can access host memory much more efficiently than coprocessor memory, just as the coprocessor can access coprocessor memory much more efficiently than host memory. To maximize performance, data that is to be accessed by the coprocessor should reside in the coprocessor memory prior to the dispatch. After the dispatch completes, the data may be moved back to host memory if it is to be accessed by the host.

Although for memory reservation and memory data moving among the two physical pools an application programming interface(API) is available, for the programmable logic configuration and dispatch steps an extra API has been developed. This API is a collection of C and assembly(platform specific) functions called by the host software. The remaining paragraphs of this sections documents the main assembly functions and their C language wrappers. For each function a brief description and its interface is given. Also a specific function has been developed in order to gather the partial results produced by each SPE.

SPECopConfig : This function is used for dispatching to the coprocessor the necessary data for its configuration registers. It requires six input parameters. The complete interface follows in table

SPECopStart : This is the function that initializes the remaining configuration registers (these not configured by the previous function) and triggers the execution of the coprocessor. Thin function return when all processing of the coprocessor has finished and the results are ready for gathering.

SPECopCall : This routine is the most general one as it combines both the configurable logic system configuration and its execution initialization.

Parameter	type	description
S1Adrr	uint64	The address of the first Stream data buffer
S2Adrr	uint64	The address of the second Stream data buffer
resultAdrr	uint64	The address of the result data buffer
resultBufSize	uint64	The size of result buffer available for each SPE
parStreamSPEs	uint64	The size of the dedicated SPEs configured in parallel and serial for each input stream
configRegsAdrr	uint64	The address for the configuration registers initial values

Table 5.3 SPECopConfig function interface

Parameter	type	description
S1Length	uint64	The size of the S1 incoming stream (in tuples)
S2Length	uint64	The size of the S2 incoming stream (in tuples)
initialIndex	uint64	the starting point for the internally generated indexes for both input streams

Table 5.4 SPECopStart function interface

gatherResults This particular function serves two different reasons, Figure 5.11 shows how this function operates. The "Main Results Memory pool" is the host memory and the "Main Memory Pool" is on the coprocessor's physical memory. It is used for moving the produced results by the coprocessor to the memory pool of the host one, but also it is used to gather these results in this memory. Particularly, because the memory allocated for the results is partitioned on 64 equally sized sub sections (as many as the SPEs), the generated results are scattered in this 64 different places in the coprocessor's memory. This function reads the first position (shown in blue) from each subsection where each corresponding SPE has stored the number of results it has produced, and subsequently copies the results starting from the second position (shown in green on the coprocessor memory) for each memory sub section to the host's unified memory, resulting in a continuous buffer of results(green on the host memory). Finally, this

function has a return value indicating the number of results generated by the whole system.

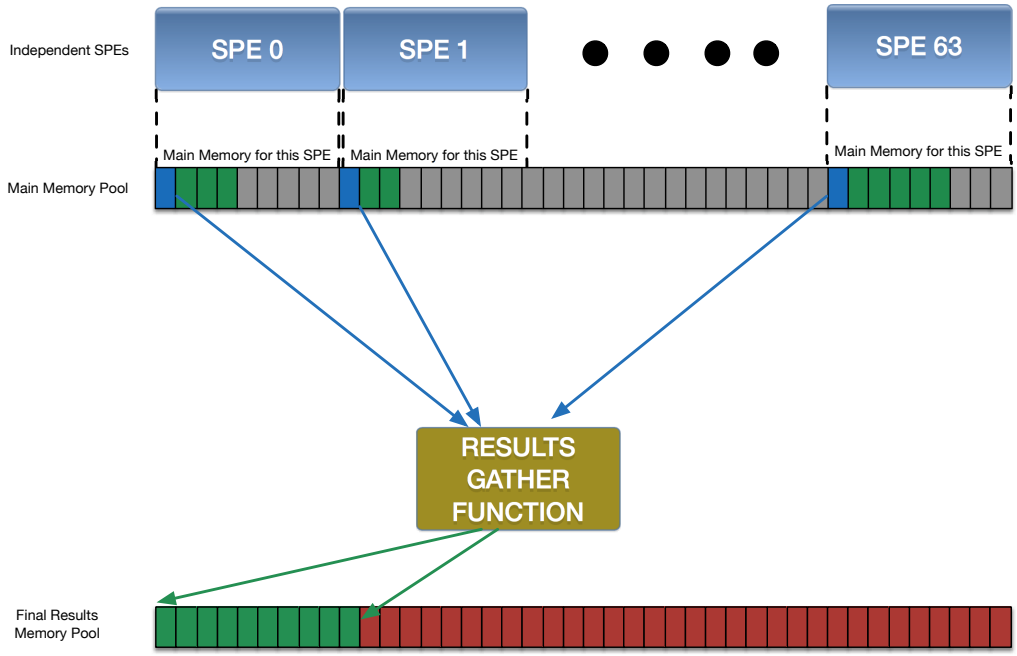


Fig. 5.11 Gather Function Operation

Parameter	type	description
cpResultsMem	uint64	The starting address of the coprocessor's memory pool
hostResultsMem	uint64	The starting address of the host memory buffer
SPEbufferSize	uint64	The size of the dedicated buffer on coprecessor's memory for each SPE

Table 5.5 SPECopCall function interface

Chapter 6

System Evaluation

6.1 Introduction

This chapter presents the evaluation of the proposed system, as described in detail in previous chapters. The first section focus on the system's verification process. The second one presents system's resource utilization and the final one a detail analysis of it's performance results with respect to the achieved processing throughput of various system configurations and where these stands in comparison with the performance of the software proposed solutions and the two previous hardware accelerated systems.

6.2 System Verification

During the evaluation process the primary target was to make sure that we have developed a completely accurate implementation concerning the output fidelity. Meaning that the output results of the proposed system are matching with the desired results. To facilitate and accelerate the evaluation process, a "gold" software reference model has been developed. This model takes the same inputs as the hardware system and generates the correct output values. These values are the reference results for the hardware implemented system. Due to the fact that the returned data from the hardware were not in the same order as the data from the reference model,one to one comparison for each returned tuple was impossible. To overcome the aforementioned problem a hash table based cross validation software was also developed. This validator inserts every result produced by the software model in the hash table, so that on a later stage each result generated from the hardware system will be hashed and checked for existents in that table. The validator also checks the number of the results produced from both the hardware system and the software model.

6.3 Recourse Utilization

In this section the resource utilization of the proposed system is presented and compared against the two previous FPGA-based solutions [4] and [5].

Table 6.1 shows that the critical resource for the proposed design is the FPGA's Look Up Table (LUTs). Although this particular resource seems not fully utilized, the further increase of the LUTs could lead in timing failures. Balancing the timing and utilization is really a difficult target. What makes this target difficult is the routing challenges analyzed in section 5.2. Moreover, Table 6.1 shows that the FPGA DSPs have been used, opposed to the previous FPGA implementations, so as to free resources on the programmable area within the FPGA and meet the placement and timing constraints.

Furthermore Table 6.2 demonstrates the significantly higher utilization achieved by this work compared with the respective utilization in both previous FPGA-based works [4] and [5]. The better utilization has been achieved after addressing all the design challenges referred at section 5.2.

Finlay, Table 6.3 shows the clock frequency achieved by all three FPGA-based systems. This work and Karandinos one [5] reached the targeted clock resulting in higher speed processing and better results fidelity than the first proposed work [4].

FPGA Resource	Available	Used	Utilization(%)
Slice Registers	948480	427340	45
Slice LUTs	474240	403661	85
Number of DSPs	864	80	9
Number of BRAM/FIFO	720	88	12

Table 6.1 FPGA Resources utilization Summary

FPGA Resource Utilization(%)	Our System	1st Proposed Solution [4]	2nd Proposed Solution [5]
Slice Registers	45	16	29
Slice LUTs	85	31	51
Number of DSPs	9	0	0
Number of BRAM/FIFO	12	15	20

Table 6.2 FPGA Resources utilization Comparison

	This Work	Kritikakis System [4]	Karandinos System [5]
Clock Frequency	150MHz	80MHz	150MHz

Table 6.3 Systems Reached Frequency Clock(MHz)

6.4 Performance Results

This section describes the performance evaluation of the implemented FPGA-based system. The performance of this system was compared against the performance of the official multi-threaded software-based system and both the previous and fastest, in comparison with the software, FPGA-based systems presented in section 4.7.

6.4.1 Theoretical bounds

The stream join processing includes the comparison of the newly arrived tuples of each one of two streams with all the in time-window tuples of the other stream. Considering tuples of both streams arriving with a rate T tuples/s and the time processing window having size W, thereafter the total number of comparisons that needs to take place at each second is about $2xWxT^2$. Thus, based on this formula we can calculate the maximum number of comparisons implemented by a system per second, given the window size and the maximum throughput rate in number of tuples per second that the system can process.

6.4.2 Experimental setup

The proposed system was developed and tested in the Convey HC-2^{ex} hybrid platform [13]. This platform as describe in section 2.6 is configured with two six-core Intel Xeon E5-2640 processors running at 2.5 GHz with 128 GB of DDR3 memory, paired with four Xilinx Virtex-6 LX760 FPGAs connected to 64GB of SG-DIMM memory.

The software-based implementation ran on a system equipped with 48 cores AMD Opteron at 2.6 GHz over 4 sockets along with 64 GB of RAM. On the other hand, the previous hardware implementations [4] and [5] were tested on the same platform as the proposed system.

The benchmark used to evaluate our system is the same used by the original ScaleJoin algorithm and many other state-of-the-art algorithms verifying the accuracy of the proposed system. Both incoming streams use random generated tuples uniformly distributed in the interval [0-10000]. Also, this kind of systems are window time-based, thus, all the systems were evaluated based on three predefined window sizes. The performance comparison of

Configuration	No of Parallel SPEs	Arrays Length
conf 64 X 128	64	128
conf 32 X 256	32	256
conf 16 X 512	16	512
conf 8 X 1024	8	1024
conf 4 X 2048	4	2048
conf 2 X 4096	2	4096

Table 6.4 Different System's Configurations Used In Our Experiments

the system used two metrics. For each metric results have been taken for various SPE configurations shown in Table 6.4. The configuration process has already described in section 5.4.

- **Tuples rate (T/s)**, which is the maximum throughput of tuples that each system can handle every second over a predefined window size.
- **Comparisons executed per second (c/s)**, which is the metric about the maximum computational operations occurring on the the incoming stream data every second, for a predefined window.

6.4.3 Processing Throughput Performance

This section presents the results of the first metric. The processing throughput achieved (in tuples per second) on the final system for various configurations is shown in table 6.4. Figure 6.1 shows that the best throughput is feasible when the system is in conf 32 X 256 configuration.

Furthermore, a comparison of the system's results and the respective multi-threaded software implementation as well as the previously proposed FPGA-based systems [4] [5], is shown in Figure 6.2. As the results indicate, the proposed solution outperforms the fastest software-based solution by a factor of up to 17$\times$ and at least a factor of about 4$\times$ the first hardware accelerated solution Kritikakis system [12]. Furthermore our system outscores the second reconfigurable solution [5] by a factor of 2.5$\times$. These results show two remarks about the new proposed hardware-based solution. Firstly, the better usage of the available memory bandwidth in the target platform can lead to at least a significant performance boost in contrast to the first previous hardware solution [12] mapped in the same platform and does not use the available memory bandwidth as effectively. Secondly,the proposed solution, due to the longer pipeline array structures can achieve better performance than both previous

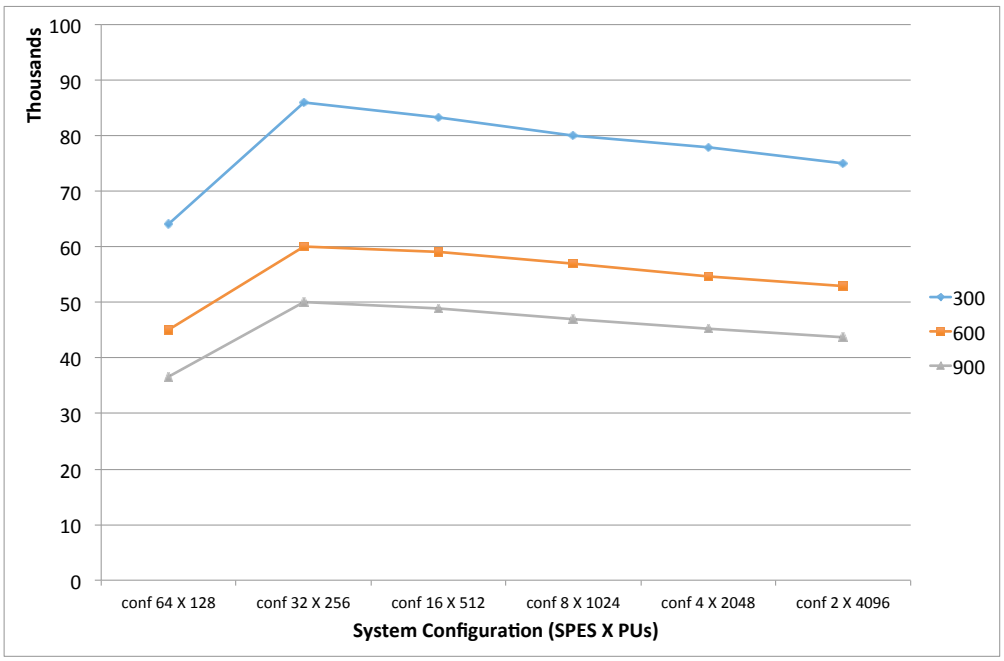


Fig. 6.1 Tuples per second for different configuration versions of the proposed system

hardware solutions. Table 6.5 summarizes the processing throughput speed up achieved by the system, undertaken during this thesis.

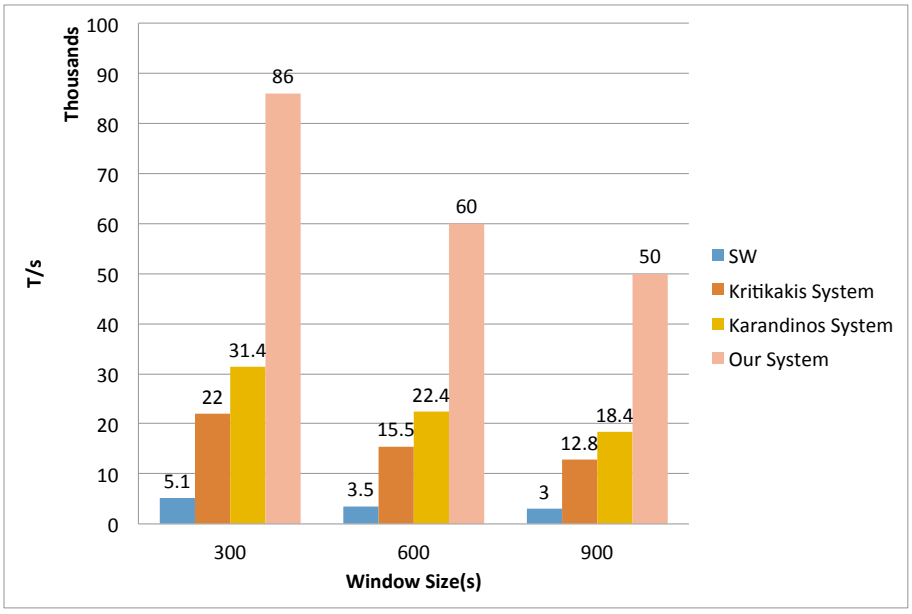


Fig. 6.2 Tuples per second for the software-based reference system and the 3 different FPGA systems

	Software System [7]	Kritikakis System [4]	Karandinos System [5]
Our System	17×	4×	2.5×

Table 6.5 Proposed Solution Processing Throughput Speed Up Summary

6.4.4 Comparisons Per Second Performance

Concerning the second metric as Figure 6.4 the best results achieved form the second configuration for all three window sizes. This Figure shows how efficient the pipeline systolic array architecture can be when it is possible to be continuously fed with data to consume.

Figure 6.3 shows the remarkable results that the proposed system performed as far as the comparisons per second metric. Software-based reference system can achieve approximately 4 billions comparisons per second for various window sizes. On the other hand, both of the previous FPGA-based systems with all four available FPGA devices can offer up to 74 and 152 billions respectively as shown in the same figure. This thesis proposed solution can reach up to the impressive numbers of about 562 billions. Thus, the proposed system outperforms in terms of the processing rate the best multi-threaded solution by a factor of about 140× when all four parallel FPGA devices are used. Also this system manage a great gains in contrast to both previous FPGA-based systems, the gain concerning Kritikakis system is 7.5× and for Karadinos system is 3.7×.

It is clear that the processing capabilities of the new-proposed system can by far exceed what the reference systems have achieved. These results prove the suitability of FPGA mapped designs for streaming workload acceleration, since these hardware devices can exploit their maximum pipeline parallelism.

6.4.5 Memory Subsystem Utilization

This section presents the measurements of the memory subsystem bandwidth utilization metric of our proposed solution for the same configurations used in previous sections. This metric is computed as the division of the number of clock cycles spent on memory subsystem requests (reads or writes) over the number of clock cycles that the system is not idle.

Although there are no results for this metric for the other two proposed solutions, this work achieves a definite higher utilization than both other FPGA-based works, since one of the requirements that we set during the design of the architecture was to exploit the available memory capabilities. This particular system keeps all available memory ports busy during the whole execution time comparing to Kritikakis solution [4] in which only two of the 16

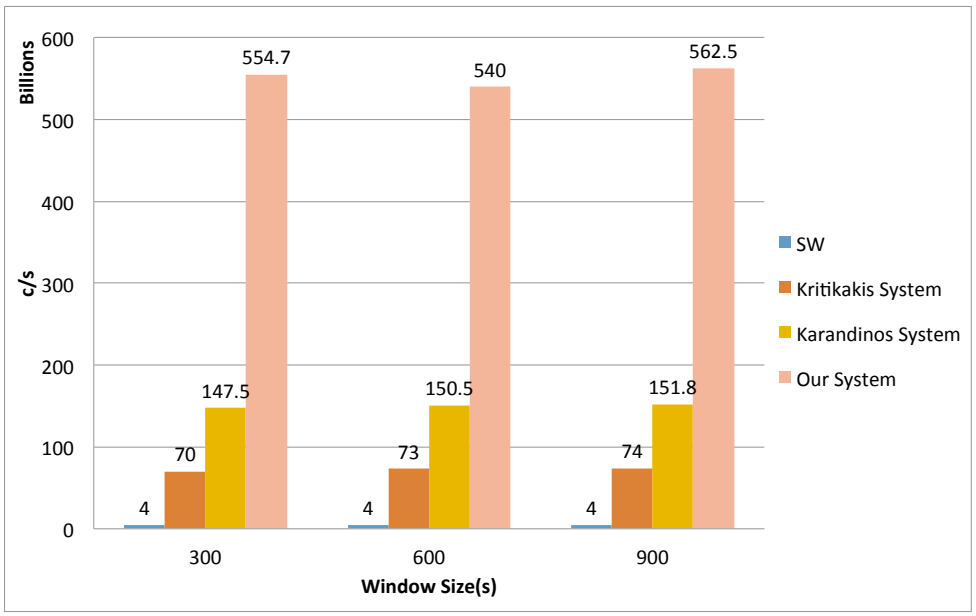


Fig. 6.3 Comparisons per second for the software-based reference system and the 3 different FPGA systems

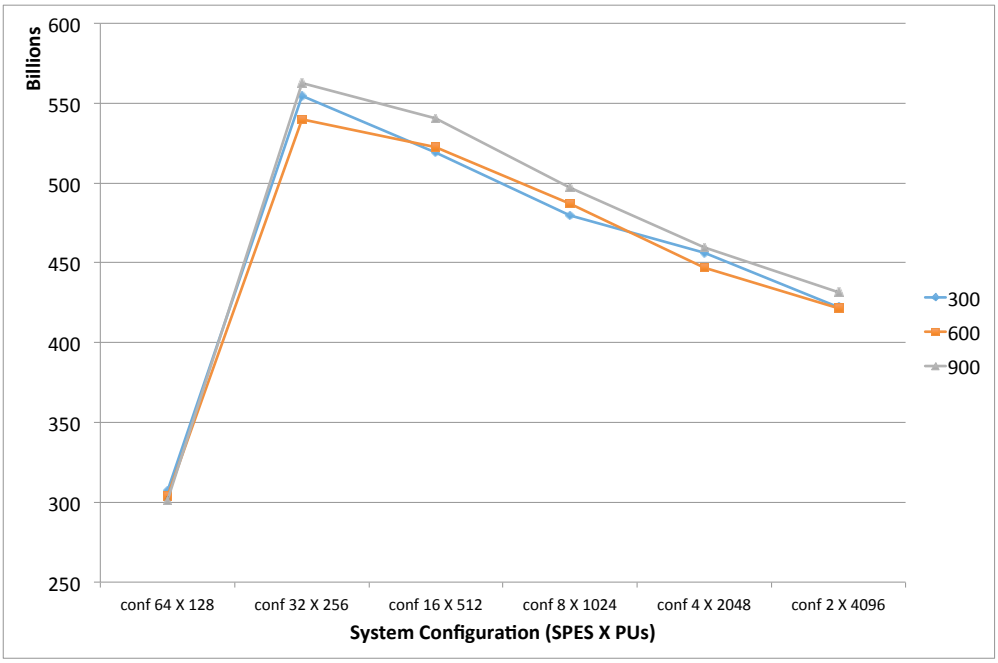


Fig. 6.4 Comparisons per second for different configuration versions of the proposed system

available memory ports per FPGA were in usage. Concerning the second work [5] although it utilize all ports its policy does not allow simultaneous operation of them. Because by the time that “new” stream data are loaded to the processing lines only two memory ports are

operating while the other 14 are waiting for the loading to finish in order to start streaming "old" stream data.

Figure 6.5 evidence that the highest utilization of the memory is achieved for the configuration that reaches the best processing throughput. The same figure shows how balanced are the requests spread among different FPGAs.

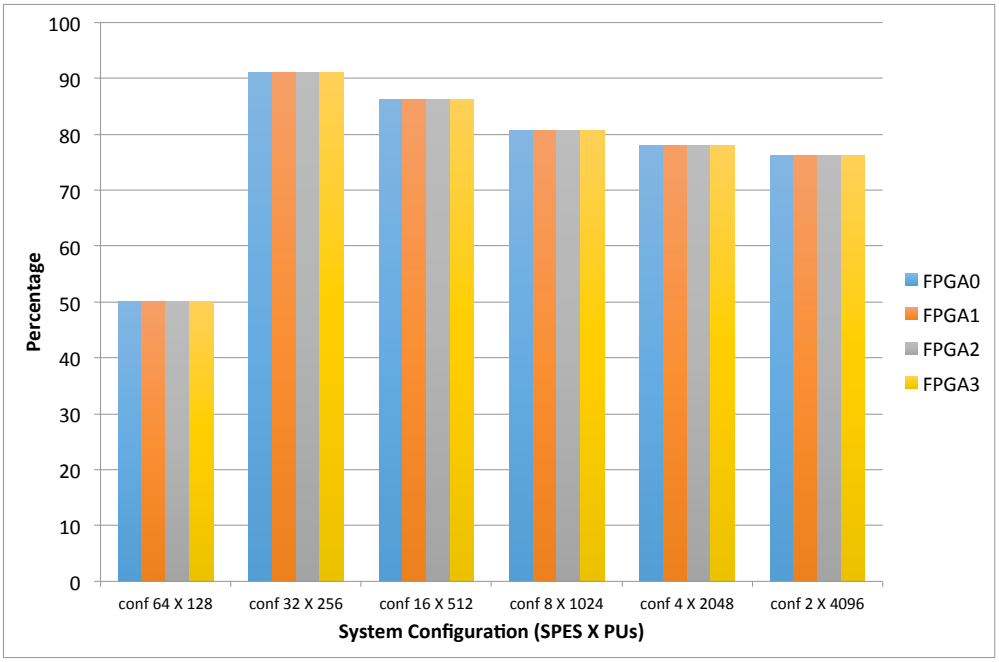


Fig. 6.5 Memory Subsystem Utilization for different configuration versions of the proposed system

6.4.6 Benchmark Performance Evaluation

The previous sections presented, analyzed and compared the performance results of the proposed FPGA-based system in comparison with the fastest software-based multi-threaded implementation of ScaleJoin algorithm. Table 6.6 compares the tuples processing throughput and the number of comparisons rate of different software-based works, the first FPGA implementation [4] its improved proposal [5] and our proposed FPGA-based system. This Table makes more clear the performance advantage of the proposed system concerning both metrics. According to Table 6.6, the throughput of our approach can be almost 10 times bigger in comparison with the best scoring software which is the Handshake implementation. This table also shows that the proposed system can offer up to 100 times higher processing rate than any other software-based state-of-the-art solutions.

System	Handshake [22],[20]	ScaleJoin [7]	Celljoin [6]	Kritikakis System [12]	Karandinos System [5]	Proposed Solution
CPU cores	40	48	9	12 CPUs & 4 FPGAs		
CPU type	2.2GHz Opteron	2.6GHz Opteron	1 PPE & 8 SPEs	2.5GHz Intel Xeon		
Peak throughput rate (kT/s)	5.125	3	2	12.8	18.4	50
Peak processing rate (Gc/s)	1.5	4	-	74	152	562.5

Table 6.6 Software and Hardware based solutions comparison table

Chapter 7

Conclusion and Future Work

7.1 Introduction

The last chapter discusses a final assessment of the design and presents recommendations for future work.

7.2 Conclusion

This master thesis presented a fully configurable and scalable FPGA-based system for accelerating stream processing algorithms. In the current work, the stream join problem, which is a vital part of stream processing and also computational intensive, was selected, implemented and analyzed as a use case. During the process of designing and implementing this system we explored and reached several conclusions about the stream processing area and how FPGAs can contribute in it. Our experimental results proved that the resulted work managed to outperform any best of our knowledge software and FPGA-based solution by a big margin in respect o processing throughput. Furthermore, the developed system also scores the largest resource utilization among all known FPGA-based systems regarding both the FPGA's internal elements and also the target platform's memory bandwidth.

In a world where the rhythm of every day data generation is exponentially increasing the need for processing data "on-the-fly" is gaining ground more rapidly than the current processing systems can serve. The performance results of this work give prominence to the capabilities of FPGAs and the recently appeared high-performance memory interfaces, which can contribute in faster and more efficient stream data processing in Big Data areas.

7.3 Target Platform knowledge Base

The development and mapping of the proposed system on the target platform required the study of the underlying hybrid (CPU-FPGA) platform, in order to make the system to reach the full potential performance. Thus, this section describes the gained knowledge during targeting to exploit both high memory bandwidth and fine-grain and coarse-grain parallelism, offered by the FPGA technology. The section presents a list of tips, split in four categories, aiming to assist next developers to achieve the best performance of their system on this platform. The first category advices regarding the best utilization of the platform's memory subsystem, whereas the second category explains the correct use of the reset and other global signals. The final two categories provide a guideline for simulation and other hints about the coprocessor and FPGA settings.

7.3.1 Memory Subsystem

- **Handling a stall from the Memory controller interface (MCIF)**

  The stall signals are driven by FIFO almost full signals, in the MCIF, so there is overrun built in to the interfaces. The custom design should stop sending requests in two clock cycles. It is expected that the stall signal is registered in the personality before it's used, and that the memory request signals are registered before being sent to the MC interfaces, resulting in two clock cycles. All requests are sent to the MC interface regardless of the state of stall, so a request so never need to resend.

- **Difference between a fence and a write flush**

  - Both the fence and the flush are used to assure the order of the accesses to coprocessor memory.
  - A fence is sent between instructions that require memory ordering.
  - The fence instruction propagates through all memory paths and ensures that all requests before the fence have been completed before any issued requests, coming after the fence sent.
  - A single custom instruction may result in memory requests issued from the developed design that need to be ordered. The design has to assert a write flush to assure write are complete before subsequent memory stores.
  - The write flush does not directly impact reads, but the custom design can wait for a flush to complete before issuing additional reads.

- **Assure proper memory ordering**
  - When writing to different coprocessor memory locations, in some circumstances write order must be assured. For example, when writing data to a buffer, then setting a valid bit. A flush must be set between the data buffer writes and the valid bit write to assure the data is written, prior to the valid bit.
  - When accessing the same memory location with a write followed by a read, a write flush is used to assure the write is complete before the read, the custom design must wait for the flush complete before issuing the read.
  - When accessing the same memory location with a read followed by a write, the custom design must wait for the read data, before sending the write, to assure the read is complete.
  - Requests should be spread evenly across the memory subsystem. The interleave spreads requests evenly across the MC's, DIMMs and banks, so sequential access or random access should perform equally well. If a particular bank is accessed more than others, then performance will suffer.
  - Bandwidth from the AEs to coprocessor memory is much higher than to host memory. the most efficient sequence for a large data structure is:
    * malloc a buffer on the host
    * malloc a buffer on the coprocessor
    * fill data structure in host memory
    * copy data structure to cprocessor memory using data mover
    * execute routine on AEs
- **Combining stall requests**

  The stall requests signals should be treated independently. Each stall request should be tied to the associated MC request logic, so only requests from that MC are stalled. Combining stall requests reduces the memory bandwidth, since a stall from one MC will stall all MC requests. It can also lead to timing problems, since all stalls must be routed to a central location where all the requests are generated.
- **Maximizing the memory bandwidth usage**
  - For maximum bandwidth requests should be generated from every AE to every MC interface on every cycle.
  - To cover the latency to coprocessor memory and achieve max memory bandwidth, it is necessary to have as many requests in flight as possible.

7.3.2 System Reset, Global Signals and Timing

- **Reasons why an application might work the first time, but not on subsequent times**

  The reset signal is only asserted during power on, not between routines. That's why the developer has to create his own reset.

- **Reset or initialize a custom design**

  The global reset signal provided by the system into the personality only asserts when the FPGA image is swapped. The personality logic will not necessarily be reset when a new program is run, so it is a good idea to reset or initialize the personality, when it is called. There's not a built-in function for resetting the personality. But the instructions are generic, so the developer could define a custom instruction to be a reset.

- **Best practices of defining timing constraints for custom designs**

  Our experience with recent versions of the tools has been that we can usually close timing at 150MHz without many constraints. Adding pipeline stages or replicating registers in the RTL usually solves timing problems. Some designs do push the technology more, and for those, adding some area group constraints can improve timing.

7.3.3 Simulation

- **Defining a simulation seed**

  If it is necessary for a simulation to be repeated the simulation seed can be defined and give the same stimulus for every execution. To achieve that n the Makefile you can define the simulation seed with either of the following environmental variables:

  - CNY_PDK_SIM_SEED
  - VERILOG_SIM_OPTIONS

- **Error message ''Number of active counters exceeds max (4)'' meaning**

  This error means that although there are four write flush counters at each even/odd MC interface a fifth write flush has been requested. Initially, a count of outstanding stores is kept using counter 0. When you send a write flush, counter 1 becomes active so that counter 0 can drain and the flush can be completed. If you send four flushes, without getting a flush complete, there are no available counters to count stores, so your

personality needs to wait for a flush complete before continuing. This error message indicates that the MCIF thinks you have sent a fifth flush.

- **simulating the memory controller interface**

  In the project testbench or sim directory, there is an sc.config file. These values are used to change the min/max response latencies. For real hardware the latency is something in the 200-300 cycle range.

7.3.4 Coprocessor and FPGA

- **keeping host executing while the coprocessor runs**

  After a coprocessor call, the host processor can continue executing asynchronously, or wait for completion of the command to be posted by the coprocessor into the status block associated with the command block. By default copcall routines wait for the coprocessor to complete. These are called blocking calls. There are versions that do not wait, which allow the host processor to continue executing asynchronously (non-blocking). A copcall with a wait is used to resynchronize, when the coprocessor completes.

- **Loading different bitfiles be on each AE FPGA**

  All FPGAs are programmed when a personality is dispatched. You must either specify a single image to program all four AEs, or you need to specify an image for each AE. The mkaetgz script constructs an AE FPGA tar archive that is suitable for installing as a Convey PDK personality image.

7.4 Future Work

Although the presented work is a great extension and improvement in comparison to the previous proposed solutions concerning the processing throughput and also the abstraction it offers, its resource utilizations does not leave much space for even more performance improvement but its functionality abstraction is still a fertile ground for future evolution. Specifically, nowadays that FPGAs are getting more and more accessible because of integrating in systems like Amazon EC2 F1 [1], which is a compute instance with FPGAs that can be programmed to create custom hardware accelerations for any application and also after Altera's (which is one of the key players in the field of FPGA) acquisition by Intel (the largest CPU supplier) [11] new hybrid systems combining commodity CPUs with FPGA are

behind the door. Such systems with an FPGA design, which will be capable of accelerating any stream processing algorithm, will soon be a necessity, and that's why a future extension of this work with a general arithmetic and logic unit integrated inside each PU unit instead of the absolute difference unit used for prototyping reasons, still holds.

References

[1] Amazon (2017). Amazon ec2 f1. https://aws.amazon.com/ec2/instance-types/f1/.

[2] Apache (2017). Apache spark. http://spark.apache.org.

[3] Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. (2002). Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM.

[4] Charalampos, K. (2016). Stream mining platform based on reconfigurable computing. diploma thesis, Electrical and Computer Engineering, Technical University of Crete.

[5] Ektor, K. C. (2017). Parallel architecture for the scalejoin algorithm implementation on the convey supercomputer. diploma thesis, Electrical and Computer Engineering, Technical University of Crete.

[6] Gedik, B., Bordawekar, R. R., and Yu, P. S. (2009). Celljoin: a parallel stream join operator for the cell processor. *The VLDB Journal—The International Journal on Very Large Data Bases*, 18(2):501–519.

[7] Gulisano, V., Nikolakopoulos, Y., Papatriantafilou, M., and Tsigas, P. (2016). Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join. *IEEE Transactions on Big Data*.

[8] Hadoop (2017). Hadoop. http://hadoop.apache.org.

[9] Halstead, R. J., Sukhwani, B., Min, H., Thoennes, M., Dube, P., Asaad, S., and Iyer, B. (2013). Accelerating join operation for relational databases with fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 17–20. IEEE.

[10] IBM (2017). Bringing big data to the enterprise. https://www-01.ibm.com/software/data/bigdata/what-is-big-data.html.

[11] Intel (2017). Intel. http://www.intel.com/content/www/us/en/homepage.html.

[12] Kritikakis, C., Chrysos, G., Dollas, A., and Pnevmatikatos, D. N. (2016). An fpga-based high-throughput stream join architecture. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pages 1–4. IEEE.

[13] Micron (2017). Micron technology, the convey hc series. https://www.micron.com/about/about-convey-computer-accelerator-products/hc-series.

[14] Miorandi, D., Sicari, S., De Pellegrini, F., and Chlamtac, I. (2012). Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497–1516.

[15] Najafi, M., Sadoghi, M., and Jacobsen, H.-A. (2016). Splitjoin: a scalable, low-latency stream join architecture with adjustable ordering precision. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 493–505. USENIX Association.

[16] Oge, Y., Miyoshi, T., Kawashima, H., and Yoshinaga, T. (2011). An implementation of handshake join on fpga. In *Networking and Computing (ICNC), 2011 Second International Conference on*, pages 95–104. IEEE.

[17] Oge, Y., Miyoshi, T., Kawashima, H., and Yoshinaga, T. (2012). Design and implementation of a merging network architecture for handshake join operator on fpga. In *Proceedings of the 2012 IEEE 6th International Symposium on Embedded Multicore SoCs*, pages 84–91. IEEE Computer Society.

[18] Oge, Y., Yoshimi, M., Miyoshi, T., Kawashima, H., Irie, H., and Yoshinaga, T. (2013). An efficient and scalable implementation of sliding-window aggregate operator on fpga. In *Computing and Networking (CANDAR), 2013 First International Symposium on*, pages 112–121. IEEE.

[19] Qian, J.-b., Xu, H.-b., DONG, Y.-S., Liu, X.-j., and Wang, Y.-l. (2005). Fpga acceleration window joins over multiple data streams. *Journal of Circuits, Systems, and Computers*, 14(04):813–830.

[20] Roy, P., Teubner, J., and Gemulla, R. (2014). Low-latency handshake join. *Proceedings of the VLDB Endowment*, 7(9):709–720.

[21] SearchCIO (2017). From data gathering to competitive strategy: The evolution of big data. http://searchcio.techtarget.com/essentialguide/From-data-gathering-to-competitive-strategy-The-evolution-of-big-data.

[22] Teubner, J. and Mueller, R. (2011). How soccer players would do stream joins. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 625–636. ACM.