Design and Implementation of a MapReduce architecture for Big Data applications using High-Level Synthesis design flow

Julian Spahr, Pedro P. Carballo and Antonio Núñez
IUMA, Institute for Applied Microelectronics, University of Las Palmas de Gran Canaria, Spain
{jspahr,carballo,nunez}@iuma.ulpgc.es

***Abstract*— This paper comprises the design and implementation of a MapReduce worker for Big Data applications on a Xilinx Zynq ZC706 using a High-Level synthesis design flow with C/C++ algorithmic descriptions. We implement a word count application such that we can estimate the main performance parameters, such as utilization, throughput and latency. We conclude the design of a DMA based MR worker with a maximum throughput of 500 Mbps considering data splitting and merging on the FPGA and 860 Mbps without, for maximum key sample of 8 KB.**

Keywords: *MapReduce; Big Data; Word Count; HPC; HLS; hardware design; Zynq*

I. INTRODUCTION

We use devices and systems that take advantage of data based infrastructure such as phones, tablets and PC's that process vast amounts of information. Examples for large datasets are streams of social networks, search engines, e-mail exchange systems, sensor networks, monitorization systems for public transport and infrastructure, vehicles such as airplanes, etc… [1], [2]. With such amounts of data, in order to provide a tolerable Quality of Service (QoS), it is crucial to understand the underlying challenges that are part of modern networks and it's data processing systems and techniques, as global IP traffic grows larger by the day [3].

The term Big Data is understood as the traffic of information that flows in immeasurable quantities between the cited sources. Ultimately, it is collected, structured, organized and indexed for its further analysis, and as the elements that compose the datasets have no value on their own, we use sophisticated algorithms, comprised in Data Mining to find recurring patterns and conceive useful information [1].

A classic approach to data processing is the MapReduce (MR) programming model, as developed by Google [4]. The MR framework, processes vast amounts of data by applying simple applications using a high amount of processing units [4]–[7]. The algorithm itself comprises two main phases: a Map phase where data is translated into useful Key-Value (KV) pairs, and a Reduce phase where the KV pairs are merged using a given criteria. Typical applications of a MR model include Matrix Multiplication, String Matching, Word Count, RGB Histogram, Machine Learning, Linear Regression, PCA, Kmeans, etc.… [5]–[7].

A MapReduce model may itself be implemented on heterogeneous platforms, as a result of requiring low-power, high-throughput applications in High-Performance Computing (HPC) environments. In order to address this need, recent research and developments propose a combination between General Purpose Processors (GPP) and Field-Programmable Gate Arrays (FPGA) on a single System on a Chip (SoC) [7]–[13]. By combining these elements on a single HPC SoC, solutions acquire the flexibility of a GPP with the high-performance and low-power response of a FPGA.

This article presents the design and implementation of a MapReduce for Big Data applications by using a High-Level Synthesis design flow on a high-end Xilinx All Programmable (AP) SoC. The employed board is the Zynq ZC706 Evaluation Board, which integrates a XC7Z045 device, which we deem enough to integrate the proposed platform. The main application under which the hardware accelerated core is designed, verified and validated is a simple Word Count application.

II. ON DESIGN METHODOLOGY AND THE XILINX ZYNQ AP SOC

It is evident that the designs performance is deeply related to its design methodology, from modelling, to synthesis and implementation of the final design. For this reason, we use the Xilinx Vivado High-Level Design Methodology, which allows for C/C++ algorithmic descriptions to be converted into a hardware IP for its later use in a hardware accelerating platform, a process we call High-Level Synthesis (HLS) [14], [15].

The underlying IP blocks for our design are modelled using C/C++ and a HLS approach. The chosen environment is the Vivado HLS tool to perform the Register Transfer Level (RTL) synthesis and the logic synthesis (implementation) [16]. The verification of the high-level design and the validation of the RTL design is made using Cadence's NCSim, as we consider it to be a more intuitive and flexible tool than the native Vivado Simulator environment.

The employed board is the Xilinx Zynq ZC706 evaluation board, which integrates a Kintex-7 derived FPGA, the XC7Z045-FFG900I device, containing 218,600 Look Up Tables (LUT's), 437,200 Flip Flops (FF's), 1,090 18 KB Block Ram's (BRAM's) (19.2 Mb) and 900 Digital Signal Processing (DSP) slices. The Processing System (PS) works at a maximum frequency of 800 MHz, while the Programmable Logic (PL) works up to a maximum of 250 MHz.

By using Xilinx provided products and design environments, we shorten valuable design and delivery time of the hardware platform. It also enables the designer to easily create hardware accelerating cores by using the Vivado IP integrator, create a bare-metal application and verify its behavior by using the Xilinx SDK environment.

III. MapReduce accelerator core

The MR hardware accelerator core is proposed as a basic MR worker, which integrates four main data processing phases: Split, Map, Reduce and Merge. By developing such a framework as a hard-IP, we create a framework whose application can be substituted with any other desired functionality. For this work, we use a simple Word Count application.

In terms of architecture, this design uses a Direct Memory Access (DMA) that connects the PS side's memory directly to the PL's custom logic, passing files from source to destination and recovering the result through AMBA AXI4 Stream interfaces (AXIS). For our purposes, we defined a MR worker by using 1 Split IP, 8 Map-Reduce pairs and 1 Merge IP. The AXIS interfaces have a width of 1 byte, since the underlying IP's do byte-per-byte processing and we reduce therefore latency derived from data type conversions. The proposed hardware accelerator core can be found in Figure I, using FIFOs to separate the individual IP blocks and create a dataflow friendly solution.

A. Split

The splitters main task is to divide the input stream into multiple streams that are sent to the available Map-Reduce lines, without destroying valuable data in the process.

B. Map

The Map IP concerns the core of the application. For our application, which is Word Count, the Map IP reads the input stream and composes Key-Value pairs, being the Key the read English word and the Value it's repetition factor. A common Map function returns KV pairs with their values as 1, since we haven't starting merging data.

C. Reduce

The Reduce function takes a KV input, compares the Key to the ones in memory, and merges it with the existing key in the registry, by increasing its value by 1. This IP requires the stream to be stored in memory, an approach that throttles greatly the throughput of the whole system. To overcome much of the introduced latency, we opted for introducing an 'index remembering' solution, that does not require reading the entire memory, but only 'legal indexes' that are allowed for any given KV pair.

For our purposes, we modelled the Reduce IP block such that it registers the second character of any given Key (e.g. for Charlie, the uppercase letter 'H') and the index in which the KV is appended or merged. If a new KV pair enters the Reduce IP, only the indexes that share that second alphabetic character is iterated. 1 letter keys are treated as 'Z' character cases, as this case of keys is underused.

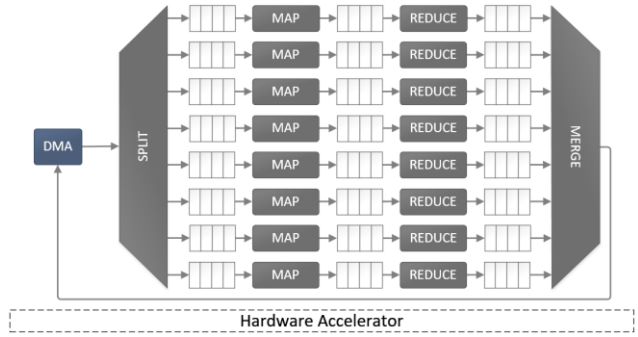

Figure I. Hardware accelerator architecture. The DMA is managed by the systems PS' and sends and recovers data from the hardware accelerator.

Once the entirety of the Key-Value pairs has been read, the memory is emptied by sending its content to the next IP block.

D. Merge

The merging function does not adhere to any application, as its sole task is combining the incoming AXI streams into one output stream that is sent back to memory through the DMA.

While the Split IP must swap interfaces after a certain number of bytes, to keep the Map stage working and not idle, the Merge IP is connected to the Reduce output and can therefore send data once it reads the first byte out of the FIFO. This means, since the Reduce stage stores the entirety of the data in memory, once valuable data is sent, the Merge IP is allowed to copy the entirety of the data to its output stream until finished, as no IP block remains idle while doing so.

IV. Modelling considerations

A. Key-Value stream protocol

Using AXIS solutions allows for easy IP block integration. However, it is crucial to develop a Key-Value based stream protocol such that the MR worker can handle the data. By a Kev-Value stream protocol, we understand a mean of transmitting data such that the IP blocks can identify the keys and values within it.

The Map IP block is the first that requires a KV specific protocol, as it translates character strings into KV pairs. For this task, we considered a KV streaming protocol. When a new key is found, it is emitted to the Reduce stage, followed by a horizontal TAB character (ASCII code 9) to mark the end of a key. Once the entire stream is processed, the Reduce IP emits the output data by sending a key followed by a TAB character and two bytes representing its value, as shown in Figure II.
For our streaming protocol, we conclude that for every TAB character on the stream reading end of the IP, we write a TAB character and two bytes of data. This results in a maximum stream size of 2 times the size of the input, which for 1 KB input depth results in 2 KB output.

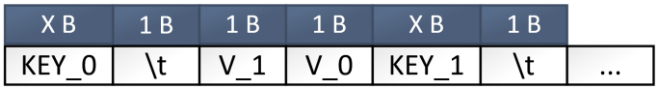

Figure II. Key-Value streaming protocol.

B. AMBA AXI4 Stream driver

We had multiple experiences using AXI Stream interfaces through C/C++ based HLS approaches. What stands out is the accessibility regarding the AXI Stream fields, which enables easy AXIS handling but carries inconsistencies within the protocol, as in other works we experienced IP block stalling due to bad stream handling. For this reason, we developed a simple C/C++ AXIS driver that handles the signals accordingly, as defined by [17]. This includes a correct handling of the TDATA, TLAST and TKEEP fields, such that standard Xilinx IP do not stall or loop indefinitely due to bad AXIS treatment from preceding IP's.

C. Platform architecture characteristcs

The designed MapReduce worker uses a 1 byte width AXIS connecting all the designed IP blocks. The accelerator has a maximum input data size of 8 KB. Although it is true that ideally the hardware accelerator can accept an indefinite amount of data as long as it is being read out of the FIFO blocks, the Reduce IP requires memory storage to happen, which therefore reduces the maximum capacity of the MR worker. We chose the Map-Reduce pairs to allow for a maximum of 1 KB, resulting in 8 KB of input data.

We decided to use a DMA based architecture, which means that the input data must be declared previously in memory and passed on to the accelerating core using the PS. We use the same DMA core to receive the resulting data. Both the transmission (TX) and reception (RX) streams are handled using interrupts in favor of polling, reducing therefore the latency derived from data transfer and handling.

D. IP block modelling

1) Split

Since we're using a Word Count application, the Split IP segments the data into 8 out-stream's. The splitting criteria is having copied a minimum of 25 bytes to the output, as well as having finished reading a complete English word before swapping to the next output interface. By doing so, the architecture permits the next phase to start working and reducing idle periods.

2) Map

The Map stage transforms the input data into valuable keys. This IP copies only the alphabetic characters from the input stream to the output, separating them with a TAB character.

3) Reduce

The Reduce IP concerns the merging of Key-Value pairs, based on their Key. An incoming Key is either appended to the KV list that is stored in memory, or merged with its existing Key by increasing its value by 1. To reduce latency derived from iterating the KV registry, the Reduce IP is only allowed to read the indexes that share a common characteristic with the current incoming Key, which for our case is the second letter. Once all the keys of the incoming stream are handled, the Reduce IP emits the entirety of its memory by using the defined KV protocol.

4) Merge

The Merge IP block reads the FIFOs located after the Reduce stage and merges the multiple streams into one output stream. If the FIFO contains valuable data, the Merge IP will read the entirety of the stream until the last element is found. While the Split IP concerns itself with swapping interfaces to establish a dataflow friendly behavioral, this stage does not need to, since finding valuable data in a FIFO means that the entirety of the Reduce result for that given line is available.

5) Platform capacity

As already mentioned, we decided to use up to 8 Map-Reduce pairs or lines after the Split IP to divide evenly the input stream and process the data in concurrently. Each line has a capacity of 1 KB, as limited by the memory of the Reduce IP block. Since we're using the KV protocol, we're outputting 2 KB for every 1 KB of data, and therefore the platform has an input capacity of 8 KB, outputting a maximum of 16 KB.

6) Implementation results

The complete utilization report is shown in TABLE I.

TABLE I. Hardware platform utilization (%)

IP	Slices	LUT	LUTRAM	FF	BRAM
Platform	31.20	20.84	30.63	6.88	32.05
Split	0.84	0.59	0.00	0.29	0.00
Map	0.78	0.47	0.00	0.26	0.00
Reduce	25.88	17.34	30.36	4.75	19.08
Merge	0.26	0.15	0.00	0.08	0.00

We conclude that the limiting factor for using a MapReduce worker is the Block RAM (BRAM) usage of the Reduce IP, which totals to 32.05 %, with the LUTRAM coming second with a 30.63 % utilization factor. We synthesize the overall utilization results, as well as the individual utilization for the multiple MR stages in Figure III.

We are also interested in calculating the Utilization Factor (UF) of our design, since, as explained in [18], the Zynq ZC706 holds a maximum of 4 LUT's and 8 FF's per slice. By calculating that factor, we are able to estimate the efficiency of the slice usage that is inferred by the HLS tool. The associated formulas are shown in (1) and (2).

$$UF_{LUT} = \frac{\text{N. of LUT's}}{\text{N. of Slices}} \quad (1)$$

$$UF_{LUT} = \frac{\text{N. of FF's}}{\text{N. of Slices}} \quad (2)$$

The UF calculations are synthesized in TABLE II, where we compare our results with Carballo's work done in [19].

Our utilization factor is on par with the overall UF_{LUT} for similar solutions on Zynq devices, as well as a Xilinx Virtex-5. None of the presented designs have a high UF_{FF}, something we can attribute to the HLS design flow and synthesis process, as well as our own design flow which does require more LUT usage over the overall FF usage.

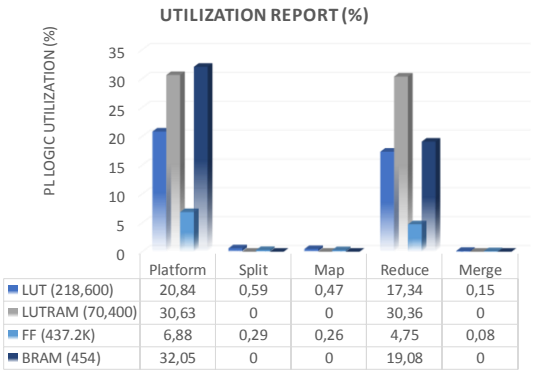

Figure III. Hardware platform utilization report.

TABLE II. SLICES UTILIZATION FACTIOR COMPARISON

Design	Architecture	HLS Tool	FPGA Device	Utilization Factor	
				UF_{LUT}	UF_{FF}
0	Platform + IP	Vivado HLS	Xilinx Zynq 7z020	2.91	2.90
1	IP	CtoS	Xilinx Zynq 7z045	3.32	1.64
2	Platform + IP	Vivado HLS	Xilinx Zynq 7z045	2.29	3.05
3	IP	Vivado HLS	Xilinx Zynq 7z045	2.12	2.79
4	Platform + IP	CtoS	Xilinx Virtex-5 FX 130t	2.85	2.09
MR Worker	Platform + IP	Vivado HLS	Xilinx Zynq 7z045	2.67	1.77

V. VALIDATION PHASE

To validate our design, we have developed a simple application that sends previously in-memory allocated data through the DMA and to the accelerator core. The result is collected by the same DMA and send back to the PS, where the KV protocol is unrolled. The given data can be either presented or stored back into memory.

A. *Measurements setup*

Our main objective in the validation phase is to validate the proper functionality of the hardware platform as a MR worker and to produce latency and throughput estimates. To do so, we used 5 samples (or keys) with a maximum size of 8 KB. Latency itself is measured using two methods:

- The worker latency from the Split to the Merge block. This method corresponds to a DMA per interruptions setup (DPIS).
- The line latency or delay associated to the Map-Reduce stage. We call this setup Reduce Per Polling (RPP) as we poll the Reduce stage to find out whether the MR worker has finished the task of processing.

B. *Latency and throughput measurements*

The obtained latencies and throughputs are shown in Figure IV and Figure V.

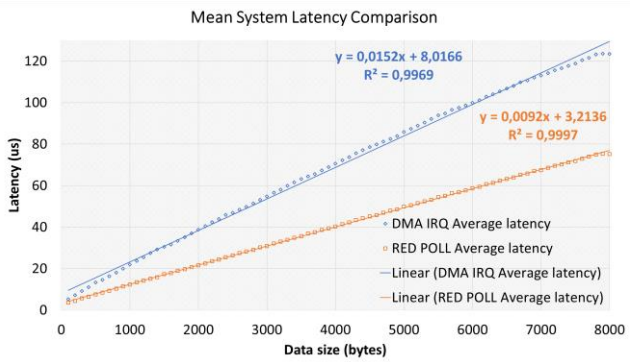

Figure IV. Mean system latency comparison.

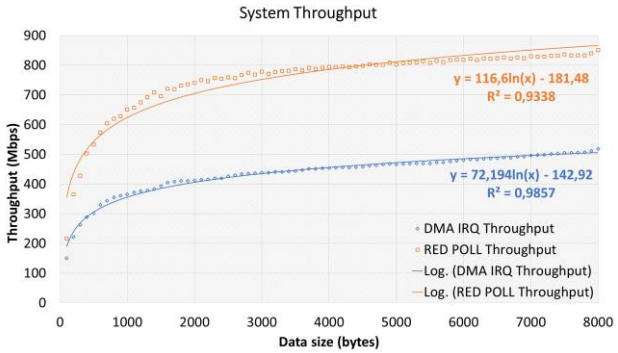

Figure V. System throughput.

We estimated the platforms parameters by calculating the mean over all the results for either of the measuring methods. The throughput measurements have a logarithmic behaviour, since the MR worker consumes a minimal latency, regardless of the input key size. We call this latency the d_{node} delay, as it is the minimal node delay. This value is of 8 µs, 3.2 µs of which belong to the RPP latency.

For a maximum sample size of 8 KB, the worker delay rises to 127 µs while the Map-Reduce pair lines have a 75 µs delay. We achieve a maximum throughput of ~860 Mbps for the RPP setup, and ~500 Mbps.

For our architecture, the test keys are previously allocated into memory, although it is possible to provide them to the ZC706 board through a Secure Digital (SD) card and automatize the working process.

VI. CONCLUSION

In this document, we present the design and implementation of a MapReduce worker platform that responds to a Big Data application on a Xilinx Zynq AP SoC. The main objective of this work is the design of the needed acceleration kernels required in hardware and in software, to implement the final MR solution.

The advantages of implementing the MapReduce worker on FPGA produces a high-throughput solution that uses the hardware acceleration kernels to increase the overall system's performance. In our case, we produce a MR worker that integrates a word count solution with a maximum throughput of 860 Mbps considering hardware data splitting and merging, and 500 Mbps in the Map-Reduce phase.

ACKNOWLEDGEMENTS

This work has been supported by the University of Las Palmas de Gran Canaria and the Institute for Applied Microelectronics (IUMA).

REFERENCES

[1] K. Adam, I. Hammad, M. Adam, I. Fakharaldien, and M. A. Majid, "Big Data Analysis and Storage," *Proceedings of the 2015 International Conference on Operations Excellence and Service Engineering*, pp. 648–659, 2015.

[2] T. J. Barnett, A. Sumits, S. Jain, U. Andra, and T. Khurana, "Cisco Visual Networking Index (VNI) and VNI Service Adoption - Global Forecast Update, 2015-2020," 2016.

[3] R. Antonello, S. Fernandes, C. Kamienski, D. Sadok, J. Kelner, I. Gódor, G. Szabó, and T. Westholm, "Deep packet inspection tools and techniques in commodity platforms: Challenges and trends," *Journal of Network and Computer Applications*, vol. 35, no. 6, pp. 1863–1878, 2012.

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Proceedings of 6th Symposium on Operating Systems Design and Implementation*, pp. 137–149, 2004.

[5] C. Kachris and G. C. Sirakoulis, "A Reconfigurable MapReduce Accelerator for multi-core all-programmable SoCs," *2014 International Symposium on System-on-Chip (SoC)*, 2014.

[6] Y. Shan, J. Yan, Y. Wang, and N. Xu, "FPMR : MapReduce Framework on FPGA A Case Study of RankBoost Acceleration," *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 93–102, 2010.

[7] D. Diamantopoulos and C. Kachris, "High-level synthesizable dataflow MapReduce accelerator for FPGA-coupled data centers," *Proceedings - 2015 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, SAMOS 2015*, no. Samos Xv, pp. 26–33, 2015.

[8] M. Aly, É. P. De Montréal, Y. Shaban, and D. Ph, "Analysis of Massive Industrial Data using MapReduce Framework for Parallel Processing," *Reliability and Maintainability Symposium (RAMS), 2017 Annual*, 2017.

[9] Z. Wang, S. Zhang, B. He, and W. Zhang, "Melia: A MapReduce Framework on FPGAs, OpenCL-based FPGAs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9219, no. c, pp. 1–14, 2016.

[10] E. Ghasemi and P. Chow, "Accelerating Apache Spark Big Data Analysis with FPGAs," *2016 Intl IEEE Conferences on Ubiquitous Intelligence Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, 2016.

[11] A. Dollas, "Big data processing with FPGA supercomputers: Opportunities and challenges," *Proceedings of IEEE Computer Society Annual Symposium on VLSI, ISVLSI*, pp. 474–479, 2014.

[12] Y. Choi and H. K. So, "Map-Reduce Processing of K-means Algorithm with FPGA-accelerated Computer Cluster," pp. 9–16, 2014.

[13] A. Cuzzocrea, M. Cosulschi, and R. de Virgilio, "An Effective and Efficient MapReduce Algorithm for Computing BFS-Based Traversals of Large-Scale RDF Graphs," *Algorithms*, vol. 9, no. 1, 2016.

[14] Xilinx, "UltraFast High-Level Productivity Design Methodology Guide," 2017.

[15] Xilinx, "Vivado Design Suite User Guide: Design Flows Overview," 2017.

[16] Xilinx, "Vivado Design Suite User Guide: High-Level Synthesis," 2017.

[17] ARM, "AMBA 4 AXI4-Stream," 2010.

[18] Xilinx, "Zynq-7000 All Programmable SoC Data Sheet: Overview," 2017.

[19] P. P. Carballo, "Aportaciones a la metodología de diseño basada en síntesis de alto nivel. Aplicaciones al diseño de IPs para procesado de eventos complejos y codificación de vídeo," 2016.