ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Grado

ACELERANDO PAGERANK CON ZCU102-ES2 FPGA

Accelerating PageRank with ZCU102-ES2 FPGA

Para acceder al Titulo de

Graduado en Ingeniería de Tecnologías de Telecomunicación

Autor: Jorge Barredo Ferreira

Julio - 2020

E.T.S. DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACION

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

CALIFICACIÓN DEL TRABAJO FIN DE GRADO

Realizado por: Jorge Barredo Ferreira
Director del TFG: Miquel Moretó Planas
Profesor Ponente del TFG: María del Carmen Martínez Fernández
Título: "Acelerando PageRank con ZCU102-ES2 FPGA"
Title: "Accelerating PageRank with ZCU102-ES2 FPGA"
Presentado a examen el día: 24 de julio de 2020

para acceder al Título de

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Composición del Tribunal:
Presidente (Apellidos, Nombre): Martínez Fernández, María del Carmen
Secretario (Apellidos, Nombre): Stafford Fernández, Esteban
Vocal (Apellidos, Nombre): Lechuga Solaegui, Yolanda

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente

Fdo.: El Secretario

Fdo.: El Vocal

Fdo.: El Director del TFG
(sólo si es distinto del Secretario)

Vº Bº del Subdirector

Trabajo Fin de Grado Nº
(a asignar por Secretaría)

Agradecimientos

A mis padres, Adrián, mis abuelos y a quien ha podido ver de primera mano cómo he trabajado e invertido tantas horas en este proyecto y a lo largo de estos cuatro años de gran esfuerzo y, sobre todo, sacrificio.

A Carmen Martínez, desde el primer día de *Microprocesadores*, que en cualquier lugar y hora me ha aconsejado cómo afrontar ese futuro que parece tan oscuro y sin luz al final del túnel, y ha dispuesto todos los medios y ayuda necesarios, más de los que un tutor común habría sido capaz de ofrecer, pudiendo encarrilar el trabajo y mi camino académico en la dirección perfecta.

Muchas gracias a Miquel Moretó, que me dio la oportunidad de poner en práctica lo aprendido en una institución del nivel y fama como es el *Barcelona Supercomputing Center*, me ha estado dirigiendo y ayudando en todo en momento y puso en mis manos la FPGA con la que todo este proyecto echó a rodar.

También a Nehir Sonmez, que me enseñó en dos meses una gran cantidad de conocimientos sobre FPGAs y sus entresijos, sin los que avanzar habría sido imposible en un mundo algo desconocido para un estudiante de Ingeniería de Tecnologías de Telecomunicación.

A Isaac Sánchez Barrera, que me dio las nociones suficientes de partición de grafos, y que serán útiles en investigaciones futuras basadas en este trabajo.

Mención al resto de compañeros del verano de 2019 del *BSC*, con los que introducirse en una ciudad nueva y tan grande como Barcelona fue mucho más fácil, y que con su experiencia he podido aprender métodos que he aplicado en este trabajo.

A todos.

Muchas gracias, Jorge

Resumen

El siguiente proyecto trata de plantear y poner en práctica un procedimiento de optimización de algoritmos de grafos ejecutados sobre una FPGA.

Para ello, se evaluarán de manera teórica los distintos problemas que aparecen en el desarrollo e implementación de los programas, tanto desde el punto de vista del hardware como del software, y se explicarán las soluciones por defecto que se han ido aplicando hasta el día de hoy.

Después se pasará a un caso práctico. Implementaremos el algoritmo de valoración de nodos comúnmente llamado *PageRank* y, tras definir su funcionamiento y una simplificación matemática, pasaremos a programarlo sin sistema operativo sobre un único procesador de la FPGA, obteniendo su tiempo base de ejecución. Más tarde, se utilizarán distintas técnicas ofrecidas por el software de *Vivado HLx* para reducir dicho tiempo de ejecución y obtener un algoritmo de grafos optimizado, de tal manera que los archivos de entrada que la placa pueda soportar sean de una escala similar a los utilizados en el mundo real.

Para terminar, pondremos en práctica una solución multiprocesador en la que no interviene ningún sistema operativo, abriendo la puerta a su posible incorporación en una investigación futura.

Palabras clave: FPGA, acelerador, algoritmos de grafos, optimización, Vivado HLx, OpenCL.

Contents

List of Abbreviations

Baremetal Computing system in which the software is programmed directly on the FPGA, not using an Operative System (OS).

BSP *Board Support Package*, includes all the software components needed to match a given operating system to a given hardware design (board) [4].

FPGA *Field-programmable Gate Array*, an electronic device containing one or more processors and many logic blocks whose functionality can be configured. It is used to execute and test software on it.

PL *Programmable Logic*, involves all the logic blocks of an FPGA: logic cells, flip-flops, DSP, etc.

PS *Processing System*, refers to the processor and peripherals of an FPGA. It can also be named as core.

WHS *Worst Hold Slack*, it is related to the worst slack of the timing paths for minimum delay analysis. If positive it means that the path passes. If negative, it fails [1].

WNS *Worst Negative Slack*, corresponds to the most negative of any single slack of the timing paths. If positive it means that the path passes. If negative, the path fails [1].

WPWS *Worst Pulse Width Slack*, refers to the maximum skew with minimum and maximum delays included [1].

Chapter 1

Introduction

Graphs are the best choice when representing data, specially in social network analytics. The way the links between nodes become more complex as its scale becomes bigger, leads us to the issue of looking for a procedure of working with this kind of inputs without spending too much resources on an optimal point of view, which is the objetive of an algorithm acceleration.

Computational acceleration consists of using software or hardware tools to execute certain program functions in a more efficient way, even more than if it was executed in a single central processing unit (CPU). This way, a computer will be able to carry out an application without spending excessive resources (from electric energy to cache and flip-flops), enabling a larger processes' concurrency and a better components' distribution.

This is why program acceleration is a **demanded issue** in this research field, and every context requires an specific development for each process.

In this chapter, the most remarkable hardware and software bottlenecks that appear when executing an algorithm without any kind of optimization will be described, as well as their default solutions that have been applied so far, ending with a group of mathematical and low-level optimizations that can be coded and used on algorithms.

1.1 Problem

A program that has not been optimized is processed directly in a CPU, without taking into account previously the maximum amount of system's resources the user wants to spend. This situation leads to a **computational overload**: the CPU queues the corresponding tasks and executes them in a FIFO (*first in, first out*) order as its frequency clock allows it to work. Besides, for every performed process, their corresponding memories and logic gates make the operation the processor asks them to do. This leads to an excessive computational waste, and the point is that every task tends to demand the highest available number of resources, in order to take advantage of all the system.

Not optimizing programs entails some bottlenecks, such as **irregular memory accesses**, and some risks related to **destructuring issues** and **poor data locality**.

1.1.1 Irregular Memory Accesses

Owing to the lack of any previous workload planning related to every system memory unit, a processor tends to send temporary instructions and data to the **cache** (indoors there can be found L1, L2, L3 levels and sometimes, an L4 level), a very fast memory that also acts as support. However, its weakpoint is its capacity, insufficient for big tasks, and that overflows very frequently.

In order to understand this problem it is needed to be aware of the memory hierarchy. *Figure 1.1* represents an overview of the memory hierarchy in the OpenCL model, described in the following lines [2].

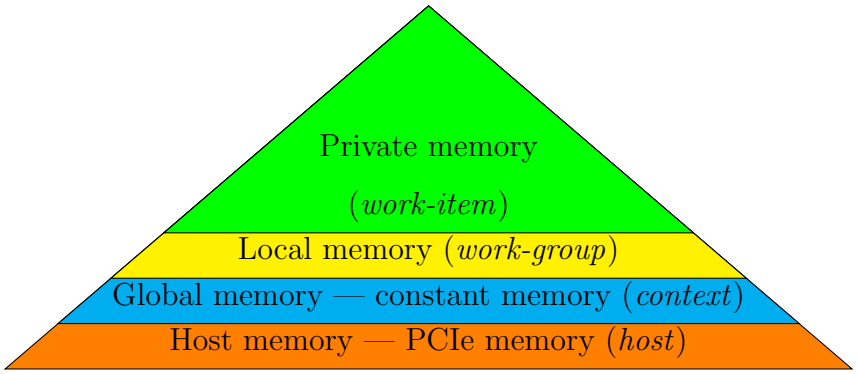


Figure 1.1: FPGA memory hierarchy

- **Private memory**: belongs to a single *work-item*, and it is not visible for the rest of them. It is the fastest one (2-3x words per cycle and *work-item*), but its capacity is too small (multiples of 10 words for every *work-item*). If used too much, data might advance to the *global memory* or the number of *work-items* that are being executed simultaneously could be reduced. It is similar to CPU registers.

- **Local memory**: belongs to a *work-group*, made up of several *work-items*. It can store 1-10x kilobytes per *work-group* and may be useful to collect necessary data for all the *work-items* it has in common, although CPUs do not have specific hardware destined for them, causing an important CPU kernels' deceleration when operated excessively. Speed of 10x words per cycle and per *work-group*.

- **Global memory**: visible for all the *work-items* belonging to the same **context**, as well as for the **host**, who can read, write and map on it. Not very stable between *work-groups*, but this effect can be reduced through synchronization. Size of 1-10x gigabytes and a speed of 100-200x GB/s.

- **Constant memory**: reading region (cache) of the *global memory* for constants initialized by the host. It has less latency[1] than cache L1 level, and contributes to reduce memory traffic inside the GPU.

- **Host memory**: host data-reachable region. It can hold 1-100x gigabytes and reaches an speed between 1 and 100x GB/s.

- **PCIe memory**: host modifiable region by the host and the graphic processing unit (GPU). Requires synchronization between CPU and GPU.

An embedded system cannot allow uncontrolled memory accesses; it requires a demanding organization to monitor every single step given on an FPGA. Based on the fact of the existence of limited resources, the cache memory should store temporarily instructions and data that are really used in a continuous way. It should also be up to save that kind of information not processed too much frequently to the biggest (and consequently slowest) memories, following the hierarchy. This way, tasks will demand less clock cycles to finish themselves, accelerating the labor of the CPU.

[1] **Latency**: amount of time that elapses from the moment an input enters a program to the obtention of its respective output from its execution

1.1.2 Destructuring Issues

When executing a random task, it is assigned to a *computing element*. However, in case of performing this task several occasions, the processor may turn **overwhelmed**, specially in those situations where the number of iterations of the inner loops is not deterministic. There should be a way to predict the number of occasions a part of the program is going to be executed, or at least, to split the work in smaller programs on the FPGA. This matter leads to a new challenge in terms of **parallelism implementation** [5].

Concurrency procedures are interesting because they give engineers the opportunity to execute different tasks at a single instance in time in different parts of the board.

Once some enhances have been applied, the speedup obtained in the new parallelized implementation can be calculated using **Amdahl's Law**:

$$SpeedUp(N) = \frac{1}{S+\frac{P}{N}}$$

Where N is the number of processors, S is the serial fraction of the code, and P is the parallel percentage from the total.

1.1.3 Poor Data Locality

Graph algorithms are often launched over graph structures that grow within a short period of time, and without any control. These memory operations show poor spatial and temporal locality features [5]:

- **Spatial locality**: existent if data is located close in memory, enabling a sequential read that saves time.
- **Temporal locality**: related to the fact that data enters the cache and leaves it later, after being used.

A non-optimized implementation may distribute data randomly throughout the memory, increasing the execution time unnecessarily.

1.2 Solution

Once unoptimized programs issues have been explained, there are two possible ways of improving timing results: implementing a low-level code that reduces the execution time only for our board, or designing a software improvement valid for any device.

1.2.1 OpenCL Proprietary Code

Coding *OpenCL* implementations is one of the most worthy solutions in order to improve algorithms' performance: the engineer works directly on hardware. Furthermore, it does not lead to a big quantity of overhead, fact that could not be possible by using other higher-level languages [6]. An *OpenCL* process program can be split in several phases, adapted to the way of working of ZYNQ devices [7]:

1. **Platform[2] and devices[3] discovery**: requires to create memory space and initialize arrays for platforms and devices, and then fill them with input data.

Listing 1.1: *OpenCL*: Platform and devices discovery

```
clGetPlatformIDs(cl_uint num_entries, cl_platform_id NULL, cl_uint 0);
(cl_platform_id*)malloc(numPlatforms*sizeof(cl_platform_id));
clGetPlatformIDs(numPlatforms, platforms, NULL);
clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, 0, NULL, 0);
clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, 0, NULL, &numDevices);
(cl_platform_id*)malloc(numDevices*sizeof(cl_platform_id));
clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, numDevices, devices, NULL);
```

2. **Context[4] creation**: the host computer sets the conditions of the working platform.

Listing 1.2: *OpenCL*: Context creation

```
cl_context context=clCreateContext(NULL, numDevices, devices, NULL, NULL,
    &status);
```

3. **Creation of a command-queue per device**: different threads for each device.

Listing 1.3: *OpenCL*: Creation of a command-queue per device

```
cl_command_queue cmdQueue=clCreateCommandQueueWithProperties(context,
    devices[0], 0, &status);
```

[2]**Platform**: specific OpenCL implementation, ie: *AMD*, *NVIDIA* or *Intel*.
[3]**Devices**: processors or working items that perform calculations.
[4]**Context**: platform that contains a pack of available devices.

4. **Creation of buffers to hold data**: it needs to be stablished if the buffer will store input data, or if it will be used to write output results.

Listing 1.4: *OpenCL*: Creation of buffers to hold data

```
1  cl_mem bufA=clCreateBuffer(context, CL_MEM_READ_ONLY, datasize, NULL,
       &status);
2  cl_mem bufC=clCreateBuffer(context, CL_MEM_WRITE_ONLY, datasize, NULL,
       &status);
```

5. **Copying the input data onto the device**

Listing 1.5: *OpenCL*: Copying the input data onto the device

```
1  status=clEnqueueWriteBuffer(cmdQueue, bufA, CL_TRUE, 0, datasize, A, 0,
       NULL, NULL);
```

6. **Creation and compilation of the program (in *OpenCL* C code)**

Listing 1.6: *OpenCL*: Creation and compilation of the program (in *OpenCL* C code)

```
1  cl_program program=clCreateProgramWithSource(context, 1, (const char**)
       &programSource, NULL, &status);
2  status=clBuildProgram(program, numDevices, devices, NULL, NULL, NULL);
```

7. **Extraction of the kernel from the program**: a program function is chosen.

Listing 1.7: *OpenCL*: Extraction of the kernel from the program

```
1  cl_kernel kernel=clCreateKernel(program, "nombre", &status);
```

8. **Execution of the kernel**: firstly, we set the buffers as arguments where data is stored. Then, it's needed to define a work-items' index space for execution, and last, the kernel is executed.

Listing 1.8: *OpenCL*: Execution of the kernel

```
1  status=clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufA);
2  size_t indexSpaceSize[1], workGroupSize[1];
3  indexSpaceSize[0]=datasize/sizeof(int);
4  workGroupSize[0]=256;
5  status=clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL, indexSpaceSize,
       workGroupSize, 0, NULL, NULL);
```

9. **Copying output data back to the host**: saved in buffer C.

Listing 1.9: *OpenCL*: Copying output data back to the host

```
status=clEnqueueReadBuffer(cmdQueue, bufC, CL_TRUE, 0, datasize, C, 0, NULL,
    NULL);
```

10. **Releasing resources**

Listing 1.10: *OpenCL*: Releasing resources

```
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cmdQueue);
clReleaseMemObject(bufA);
clReleaseMemObject(bufC);
clReleaseContext(context);
```

1.2.2 Algorithm Coding Optimization

If developing a specific OpenCL code for an algorithm is not possible, there are more available options in order to improve performance, which require to start the code from the beginning and to know very well the hardware architecture that is being used.

1.2.2.1 Algebraic Approximation

It has been shown that computing savings are related to the way an algorithm is mathematically resolved. It is never implemented directly, but optimized by hand by specialists such as mathematicians, physicists or even engineers.

In this project we will face **undirected graphs**, which is more suitable to work on than directed ones. This issue is common in the **breadth-first search (BFS) algorithm**, which is very useful when travelling through a graph [8]. As its name says, it moves "horizontally", and finds the shortest path between the nodes of the graph.

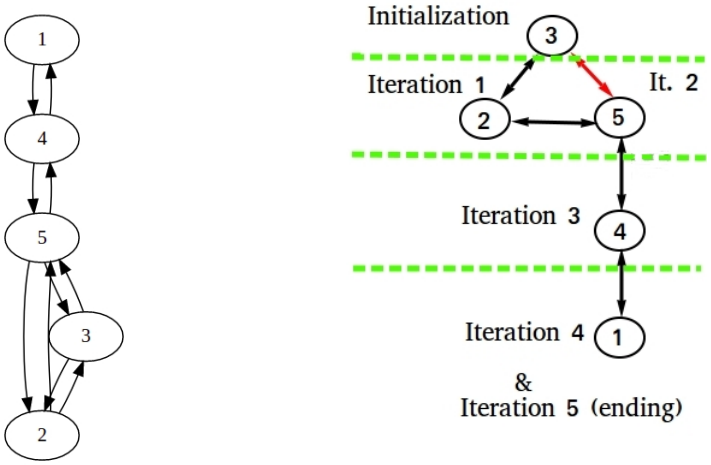


Figure 1.2: Sample graph (left), BFS "handmade" resolution (right)

Considering the sample graph [5] in *Figure 1.2*, and the fact that **node 3 needs to know the shortest path to the rest of nodes**. An engineer who needs to solve this issue would initialize *node 3*, discard it and move to one of its neighbours, discard this one, move to another neighbour, and so on, until the last node is reached. In *Figure 1.2* there is the "handmade" solution, **made after six iterations** -including initialization and ending.

This kind of solution can be done by hand, but what if a hardware implementation is demanded? Then we would think about **dynamic memory allocations (*mallocs*)**. In fact, that is how this algorithm is coded in university degrees. On the other hand, it is known that those techniques are very relevant when performing on powerful PCs, but not on all FPGAs and other devices. A different choice should come into play. That is the moment when **algebraic approximations** appear [9].

Foremost, a graph is usually conceived as a matrix (AM, *adjacency matrix*), where each value is "1" if a link exists and "0" if not. The row index represents the origin node and the column value the destination node. For example, if *Matrix(3,2)=1*, then there is a link from node 3 to node 2. In undirected graphs, this matrix will be symmetric.

We can obtain the BFS dealing with the matrix thanks to an *initialization vector* (IV), a column vector. Its number of rows corresponds to the number of nodes of the graph, and *IV(initialization node,1)=1*. The result of AM*IV is another column vector full of zeros, but, not casually, with a "1" in the rows/nodes which are neighbours of the initialization node (which is *node 3* in this example).

[5] Graph made with Graphviz

If we multiply AM and the last result, we get another column vector. It is needed to **repeat this step until the column vector does not have any zero**. However, every step can overwrite values in the results column vector. That is why it has to be declared another column vector (*dist*) where we will add the number of step to the positions of the **recently discovered nodes**.

- **Step 1 (initialization)**

$$y = AM * IV = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix} * \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \qquad dist = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

This means that neighbours for *node 3* are *node 2* and *node 5*.

- **Step 2**

$$y = AM * y = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix} * \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 1 \\ 1 \end{bmatrix} \qquad dist = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 2 \\ 1 \end{bmatrix}$$

Node 4 discovered in step 2.

- **Step 3**

$$y = AM * y = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix} * \begin{bmatrix} 0 \\ 1 \\ 2 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 2 \\ 1 \\ 4 \end{bmatrix} \qquad dist = \begin{bmatrix} 3 \\ 1 \\ 0 \\ 2 \\ 1 \end{bmatrix}$$

Node 1 discovered in step 3.

It can be seen that, if we try to solve BFS by hand, we need six iterations (*number of nodes + 1*), but **thanks to the algebraic approximation, the number of iterations reduces to three**. This is due to the fact that **handmade BFS advances node per node, and algebraic approximation allows to advance neighbourhood by neighbourhood** (represented as discontinuous green lines in *Figure 3.2*).

This leads now to a worse computing cost (many matrix products), but if we implement this to a large-scale graph we will find there is a computing benefit. In conclusion, algebra makes things easier for coding algorithms [10].

1.2.2.2 Compiler Flags

When coding, it is important to use this kind of indications to the compiler in order to optimize the execution time. There is a huge variability of them, but the most important ones in terms of computing performance are these below [11]:

- ***-O0, -O1, -O2, -O3, -Ofast***: the most common flags. Each one involves big groups of recommended indications, making it easier to optimize any program.
- ***-ftree=loop-vectorize***: tries to impose vectorization when a task is executed several times, including loops.
- ***-pipe***: tells the compiler to avoid temporary files, speeding up builds.
- ***-funroll-loops***: asks for unrolling loops whose number of iterations can be determined at compile time or upon entry to the loop. It also exists a flag (***-funroll-all-loops***) that works in the same way and unrolls those loops even when the number of iterations is uncertain before entering the loop, fact that would deal to slower programs.

1.2.2.3 HLS Pragmas

The HLS tool included in Vivado HLx software provides support for pragmas that can be used to optimize designs in terms of decrementing area usage, rising throughput performance, reducing latency... They can be added as a directive (implemented but not visible along the RTL code) or as a line of code to indicate directly which part of the program will be optimized. The most common pragmas are the following ones [12] [13]:

- **Array-reshape**: combines array partitioning with vertical array mapping.
- **Dataflow**: enables task-level pipelining (functions and loops overlaping). Without other directives, HLS seeks to minimize latency and improve concurrency.
- **Inline**: dissolves a function into the calling function.
- **Latency**: specifies a minimum or maximum latency.
- **Loop flatten**: allows nested loops to be flattened into a single loop hierarchy with improved latency.
- **Loop merge**: merges consecutive loops into a single loop to reduce overall latency.
- **Loop tripcount**: specifies the total number of iterations performed by a loop manually.
- **Pipeline**: reduces the initiation interval for a function or loop by allowing the concurrent execution of operations.
- **Unroll**: similar to the compiler flag.

1.2.2.4 Memory Reorganization

When discussing about data locality, there is a solution coming to our minds: information and instructions should be packed together in a location where memory works faster. This issue is related to several parameters that are ought to be changed once this point is reached:

- **Data and instructions location**: the most suitable places of the FPGA are the RAM and cache memories. As well as their capability is the smallest compared to other existent structures, it is enough for some large-scale graph algorithms, and they are the fastest ones, so **they will belong to the baseline of this project**.
- **Stack memory**: in charge of temporary static variables, it holds very fast access and works as a LIFO. Graph algorithms tend to make use of the same node many times (the number of links with it as origin or destination node), so it's dropped off and taken back from stack memory continuously. **It's required to increase its size in order to face bigger graphs**.

- **Heap memory**: manages dynamic memory allocation. The elements of the heap does not have dependencies with each other, and they can be accessed at any time. Taking advantage of the FPGA, **its size will also be grown**.

1.3 Document Structure

The practical part of this project consists of several chapters:

- ***Experimental environment***: in this chapter, hardware and software are described, followed by the baseline procedure that will be optimized in the next chapter.

- ***PageRank Acceleration using a ZCU102-ES2 FPGA***: this block explains how two implementations are built, one based on a single-core and a multicore solutions.

Chapter 2

Background and Experimental Environment

After discussing theoretical problems in algorithms' optimization and possible fixes that can solve them, we will build a procedure in order to improve the execution time of the programs and reduce the number of components required to run them.

Firstly, the used hardware and software suites will be described, as well as the possibilities they offer, and the algorithm we are going to enhance. Then, we will follow a procedure over the three programs considered in this project, obtaining a baseline implementation that will bring back a execution time and components' usage that we will take as reference for the optimizations that will be applied in the next steps.

Furthermore, after the hardware and software designs have been improved, it will be detailed a multiprocessor baremetal solution that can be taken into account when implementing an Operative System on the FPGA.

2.1 Hardware

In this project, we will make use of the **Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit**. Graph algorithms will be executed in this board, as well as different configurations in order to analyze the evolution of the performance while improving the design. It includes:

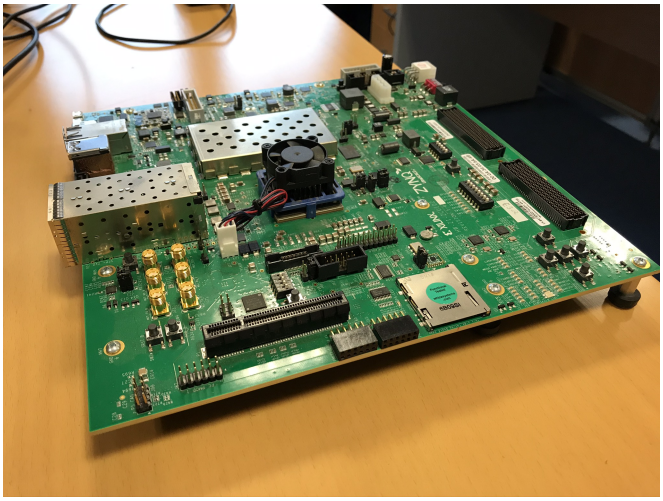


Figure 2.1: Hardware used in the project

- Processor *Zynq UltraScale XCZU9EG-L1FFVB1156I-2i-ES2*, consisting of:
  - *Quad-Core ARM Cortex A-53* (Appendix A) with several levels: L1I (32KB/-core), L1D (32KB/core) and L2 (128Kib-2MiB/cluster).
  - *Dual-Core Cortex-R5F* CPU.
  - *Mali-400 MP2* GPU.
  - 600K system logic cells.
  - 32.1 Mb of memory.
  - 2520 DSP slices.
  - Maximum of 328 I/O pins.
- PS 4GB RAM DDR4-2666, with a working speed of 21.33 GB/s max.
- PL 512MB DDR4 component memory.

Appart from the FPGA, it will be used as host computer a **Lenovo Legion Y530-15ICH**, with an Intel Core i7-8750H, 8GB RAM, 512GB SSD and powered by *Windows 10 Home*, v1909. *Ubuntu 16.04 LTS* was also tested for this project, but it was finally abandoned due to issues related to drivers.

2.1.1 Ports in ZCU102-ES2

This FPGA includes several ports that can be used to introduce input information for the different applications. It can be distinguished four categories [3]:

- **UART (*Universal Asynchronous Receiver-Transmitter*)** (*Figure 2.2*): used for asynchronous serial communication over the computer serial port. In this project, it will be used to send commands and receive timing results from the FPGA, with an adjusted speed of 9600 bits per second.

- **JTAG (*Joint Test Action Group*)** (*Figure 2.2*): conceived as a hardware interface that allows a computer to debug, program and test embedded devices. It works as a master/slave interface, and it is where the algorithm code will be sent to the FPGA through.

- **Ethernet port** (*Figure 2.2*): *ZCU102-ES2* implements a 10/100/1000 Megabits per second Ethernet interface, routed to an RJ45 Ethernet connector. It is linked to the physical address **5'b01100 (0x0C)**.

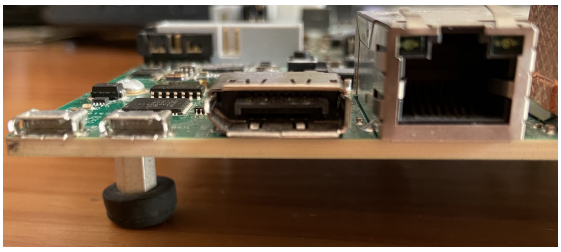


Figure 2.2: UART, JTAG and Ethernet ports disposition over the FPGA

- **SD-card port** (*Figure 2.3*): provides access to non-volatile memory cards and peripherals for I/O applications. In this project, we will insert through it the input graphs that the FPGA needs to execute PageRank. The chosen memory card is a *SanDisk Ultra SDHC Memory Card Class 10 FFP, 16 GB* with up to 80 MB/s.

Figure 2.3: SD-card port position on the FPGA

- **USB port**: not used in this project.

2.2 Software and Programming Languages

As well as it is going to be used a Xilinx-based FPGA, graph acceleration will be configured through tools developed by this company, specially three ones included in **Vivado HLx software suite**. We will use them in **version 2017.4**:

- **Vivado HLS**: converts a C, C++ or SystemC design specification into Register Transfer Level (**RTL**) code, which could be used in other Vivado tools as part of a logic design (**HLS IPs**) in an integrated circuit design. In this project, it will be important to **design hardware blocks based on HLS directives or pragmas**, included in a C program where PageRank is defined.

- **Vivado**: allows the user to build his own hardware design from zero, taking as reference the considered FPGA hardware template. We will use it to specify which FPGA components are needed and the memory addresses that the program is going to use, appart from **testing its timing and power**.

- **Xilinx SDK**: based on *Eclipse IDE* and working with Vivado hardware designs, it enables the user to create software platforms and applications targeted for Xilinx embedded processors [14]. Basically, this is the IDE that imports a **BSP** (including a bitstream file) from a Vivado project and where PageRank algorithm will be programmed into the FPGA (**in C++ language**), and where we will be able to change how much quantity of heap and stack will be used and in which component both will be placed. It is also important when choosing which processor execute each computing task, in issues related to **computing parallelization**.

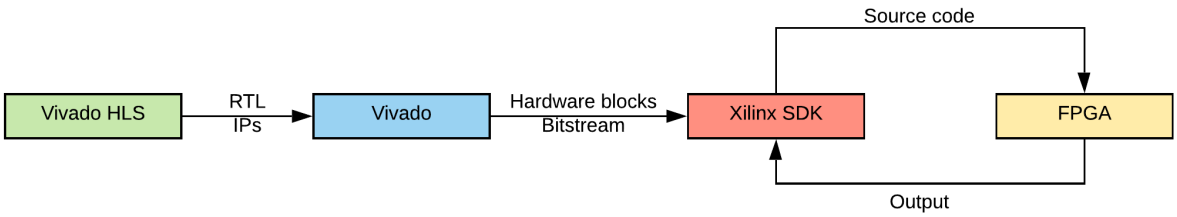


Figure 2.4: Software functionality scheme

2.3 PageRank

This algorithm was developed in 1998 by Google's founders Serguéi Brin and Larry Page, as a solution for those websites that had an important relevance for Internet users, but due to the increasing expansion of Web's size and the massive work of spammers, they did not appear on the first pages when performing a search query.

Google's PageRank way of working revolves around a popularity score system that rates every website, taking into account if it is pointed to by other important pages. These scores are displayed as an integer from 0 to 10, representing the "importance" of each website in relation to the whole Internet graph, and they are updated as well as users visit that page. This search resolution is still being applied nowadays.

This algorithm is executed iteratively. All the vertex scores are calculated along every iteration thanks to the result of the update of their neighbours, whose value is made up of the next equation [15]:

$$PR_{i+1}(v) = \frac{1-d}{|V|} + d \sum_{u \epsilon N_i(v)} \frac{PR_i(u)}{|N_0(u)|}$$

An implementation coded in *C* language is described on *Appendix B*.

2.4 Baseline Implementation

The basic process we have to follow in order to program an algorithm on the FPGA consists of three phases that will be repeated several times, even when we want to optimize designs. The procedure corresponds to the steps related in *Figure 2.4*.

1. **Designing blocks in Vivado HLS**: needed if the engineer needs to "book" a explicit and fixed number of components that will be executed on the board, as *IP* hardware blocks.

2. **Building an FPGA hardware design in Vivado**: where a board design is loaded. The previous created *IP* can be added to this design, that will be exported to *Xilinx SDK*.

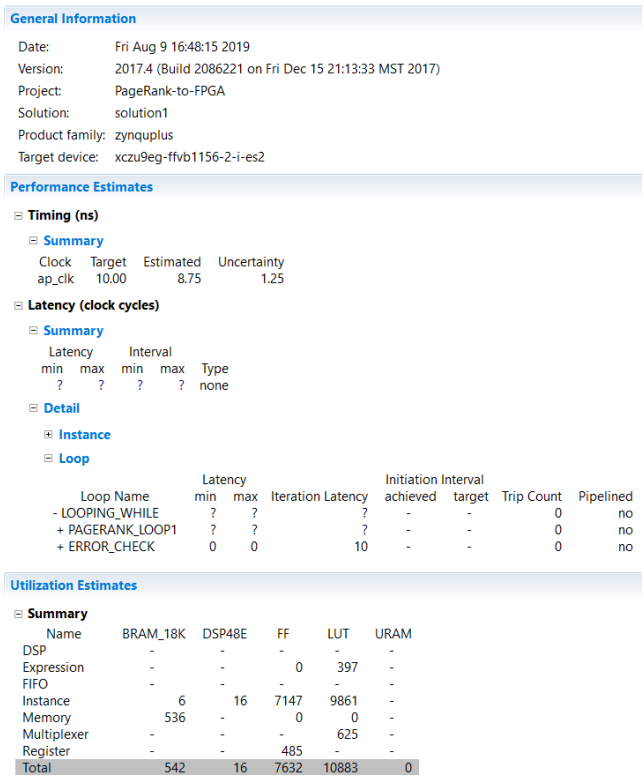
3. **Using Xilinx SDK to program the FPGA**: after importing the hardware design to the *SDK*, the algorithm will be coded here, as well as the desired optimizations, libraries... Finally, these changes will be saved and programmed on the board.

2.4.1 Designing Blocks in Vivado HLS

Vivado HLS manages the creation of hardware blocks that can be exported as part of Vivado designs. These hardware blocks are compiled after coding the algorithm or program the FPGA is going to execute, working as a kind of token that books the exactly quantity of required resources to run the program. The operation mode of *Vivado HLS* consists of two kind of files:

- **Source files**: containing the algorithms the FPGA will be able to perform. The source files can hold ***pragmas*** and memory instructions such as ***callocs***, allowing several types of optimizations that direct hardware orders do not support.
- **Testbench files**: including the input data and functions calling, they are created with the objective of testing the latency, timing, resources estimates... When executing the program, one testbench file will be selected.

After defining different files, selecting a main function to test and running the program, *Vivado HLS* will bring back a *Synthesis Report* (*Figure 2.5*) that explains the function's performance (timing and latency results), how much and which kind of resources have been involved, and which type of interfaces have been used. It allows the engineer to find out how optimizations affected on the original code.



General Information

Date:	Fri Aug 9 16:48:15 2019
Version:	2017.4 (Build 2086221 on Fri Dec 15 21:13:33 MST 2017)
Project:	PageRank-to-FPGA
Solution:	solution1
Product family:	zynquplus
Target device:	xczu9eg-ffvb1156-2-i-es2

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.75	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
?	?	?	?	none

Detail

Instance

Loop

Loop Name	Latency min	Latency max	Iteration Latency	Initiation Interval achieved	Initiation Interval target	Trip Count	Pipelined
- LOOPING_WHILE	?	?	?	-	-	0	no
+ PAGERANK_LOOP1	?	?	?	-	-	0	no
+ ERROR_CHECK	0	0	10	-	-	0	no

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	397	-
FIFO	-	-	-	-	-
Instance	6	16	7147	9861	-
Memory	536	-	0	0	-
Multiplexer	-	-	-	625	-
Register	-	-	485	-	-
Total	542	16	7632	10883	0

Figure 2.5: Fragment of *Synthesis Report*, from Vivado HLS

2.4.2 Building an FPGA Hardware Design in Vivado

Firstly, in **Vivado**, we chose an *Example Project* provided by the program for our FPGA (*Figure 2.6*), suitable for general purposes, and synthesized it in order to get theoretical power and component use results (*Table 2.1* and *Table 2.2*).

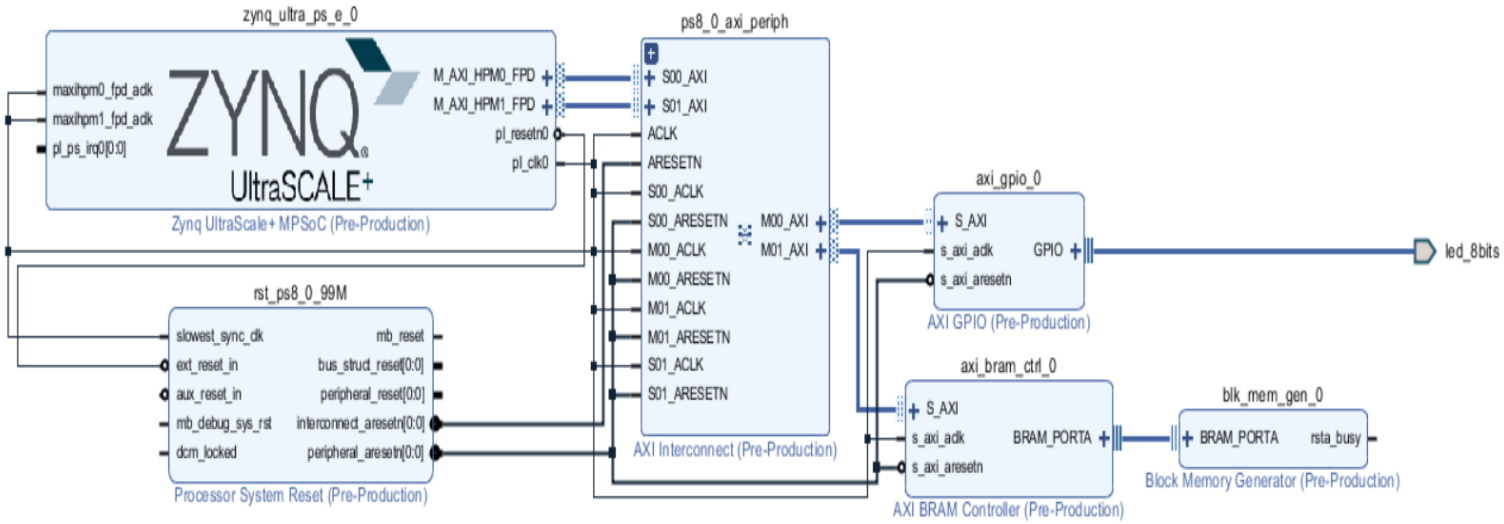


Figure 2.6: Block design of the *Example Project* in Vivado

On-chip power	4.977W	*Dynamic power*	4.214W (85%)	*Clocks*	0.196W
				Signals	0.106W
				Logic	0.205W
				BRAM	0.071W
				DSP	0.015W
				PLL	0.059W
				MMCM	0.114W
				I/O	0.253W
				PS	3.194W
		Static power	0.763W (15%)	*PL*	0.663W
				PS	0.100W

Table 2.1: Windows basic implementation on-chip power

Element	*BRAM_18K*	*DSP48E*	*FF*	*LUT*	*URAM*	*WNS*	*WHS*	*WPWS*
Quantity	297	19	26948	26719	0	0.628ns	0.01ns	0.058ns

Table 2.2: Windows component usage and critical path stats

As it was said before, *Ubuntu 16.04 LTS* was also tested as host OS for Vivado projects. When building the same block design as the previous implementation in Windows, results were much better (*Table 2.3* and *Table 2.4*). However, some driver problems of the laptop showed up and Windows was finally set as main Operative System.

On-chip power	3.934W	*Dynamic power*	3.209W (82%)	*Clocks*	0.008W
				Signals	0.006W
				Logic	0.005W
				BRAM	0.001W
				DSP	0
				PLL	0
				MMCM	0
				I/O	0
				PS	3.190W
		Static power	0.725W (18%)	*PL*	0.626W
				PS	0.099W

Table 2.3: Ubuntu basic implementation on-chip power

Element	*BRAM_18K*	*DSP48E*	*FF*	*LUT*	*URAM*	*WNS*	*WHS*	*WPWS*
Quantity	0	0	2318	2454	0	5.110ns	0.034ns	3.498ns

Table 2.4: Ubuntu component usage and critical path stats

Nevertheless, we included an specific PageRank hardware block (previously designed in Vivado HLS) in the scheme as a way to compare the performance when it is used or not. The next step implies to compile the project, getting its ***Board Support Package* (BSP)** and its **bitstream file**, that will be exported to **Xilinx SDK** attached to a defined hardware configuration. Those are the minimum files required for programming the project on the FPGA.

2.4.3 Using Xilinx SDK to Program the FPGA

Once we enter the program, we will create a new application project in SDK. This project contains **source files**, where we will code the algorithm (Appendix B) and a small graph textfile (with a size of more than 450 nodes and 2025 edges[1], ie: *Figure 2.7*); and the previously mentioned **BSP**, whose content includes a key file: ***system.mss***. This file alludes to the target board and target processor for the project, its Operative System, which peripheral drivers are being made use of, and last but not least, it is possible to choose **libraries**, in charge of enabling some special features such as a Fat File System, Power Management APIs, flash memories...

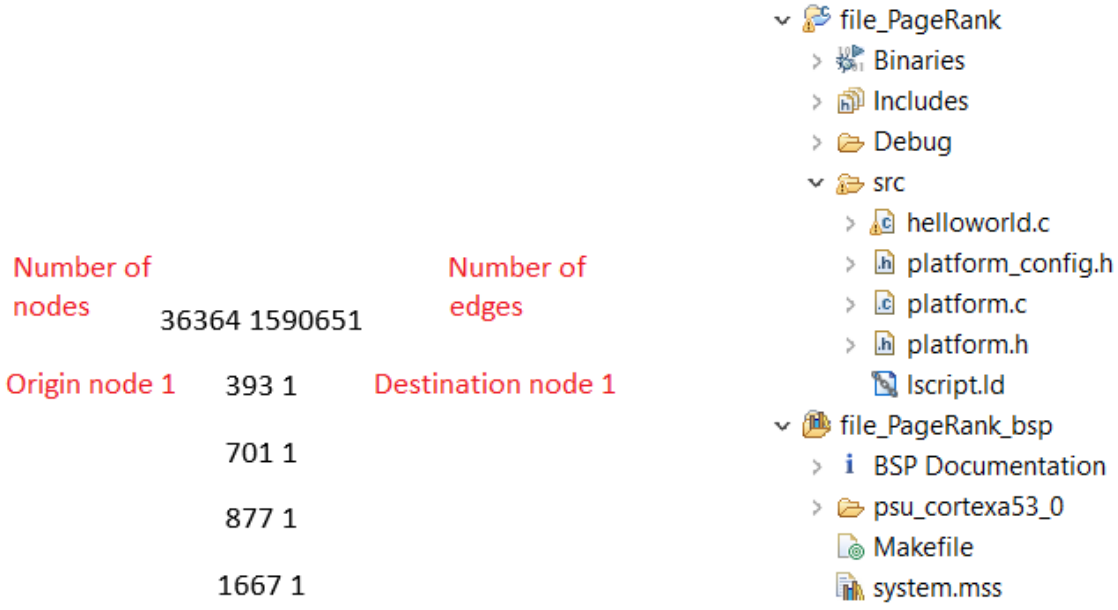


Figure 2.7: Graph textfile sample (left), Xilinx SDK project files hierarchy (right)

In *system.mss* it is also required to enable *libmetal* and *xilmfs* (Xilinx Memory File System) libraries, used in baremetal implementations and containing files related to file system applications respectively.

Before programming software on the FPGA, **compiler flags** or **linker flags** can be imposed when building the project, depending on the requirements of the program itself. It can be done by right clicking on the project folder and clicking *Properties* (*Figure 2.8*).

[1]Edge: refers to a link between two nodes.

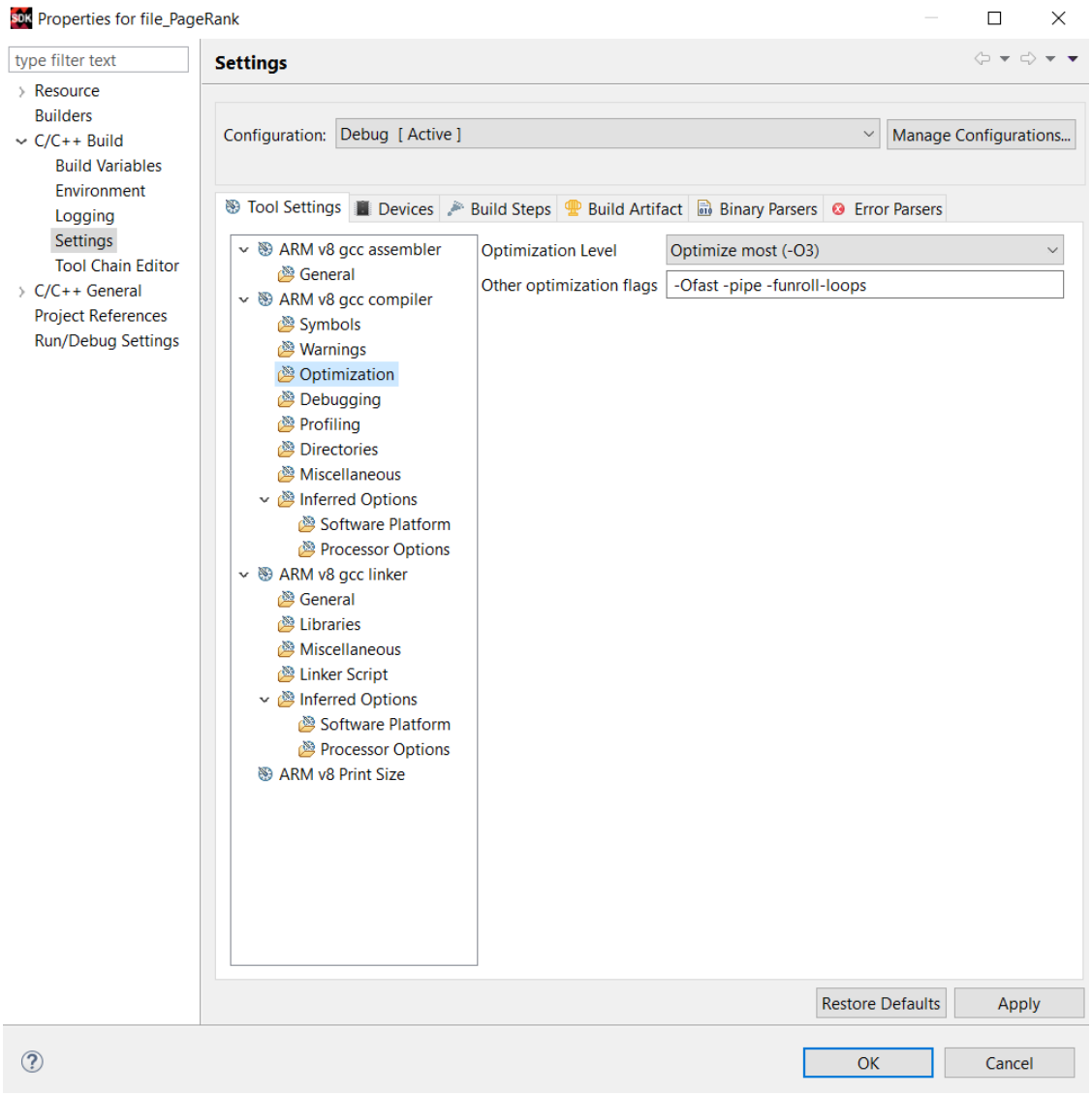


Figure 2.8: C/C++ Build Settings in properties window

After this steps, it is time to program the project on the FPGA, setting **COM3** as the host laptop port. Then, we execute each compiler flag configuration ten times (right click on project folder, click *Run As* and *1 Launch on Hardware (System Debugger)*) and we calculate the average. Henceforth, timing results tables will show the time that lapses from the calling to the PageRank algorithm function to the instant it ends.

In *Figure 2.9*, they are compared the design that includes specific hardware block and the one that it does not (x-scale: implemented compiler flags, y-scale: execution time). It can be seen that the time difference between inserting and deprecating the hardware block is minimal, as well as using more compiler flags results on a smaller execution time. That is the reason why this block is not useful in our project, and will be not be used in the optimization that will be applied later.

Finally, this chart also shows how the most restrictive compiler flags do not always bring the best timing results, as it can be seen on the green bars. We will use a more diverse repertory of flags that will impact on a smaller execution time.

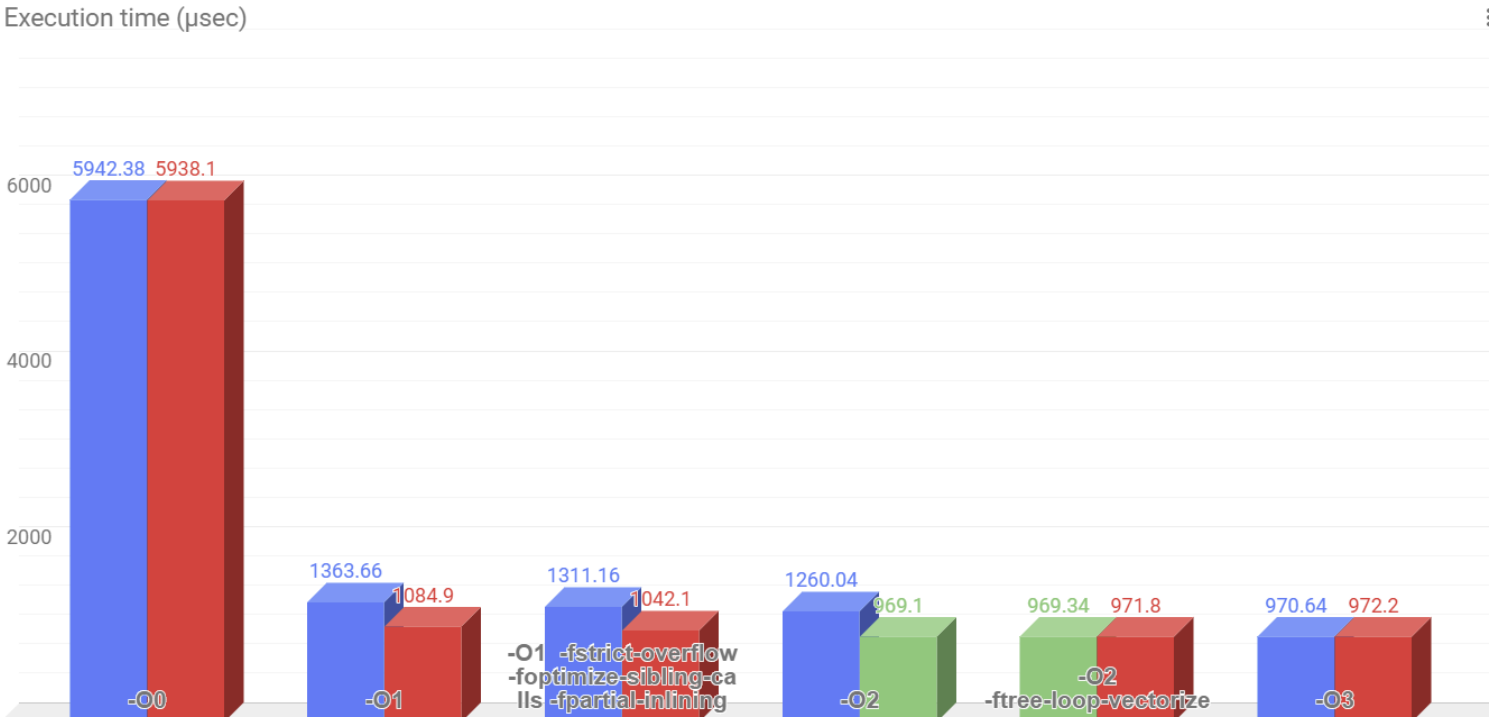


Figure 2.9: Timing results comparison (with and without PageRank hardware block)

Chapter 3

PageRank Acceleration using a ZCU102-ES2 FPGA

Once the basic PageRank disposal has been settled, it is time to build an improved implementation. Based on the theoretical ideas and the procedure followed before, monocore results will be obtained after implementing an SD card driver. Then, stack and heap memories will be redistributed and their size adapted thanks to the introduction of a *linker script*, ending with the execution of large-scale graphs that will try out the capability of the *ZCU102-ES2*.

Last but not least, it will be executed a **multicore baremetal development** that has been tested, leaving some interesting results that can be considered when implementing an Operative System on the FPGA in future research.

3.1 Single Core Performance

Monocore execution procedure consists on calculating the timelapse when inserting graphs as a textfile in the SD card, and from that point a *linker script* will be configured and added to the board. It will show hefty diferences that will make us reflect on how memory accesses act as a bottleneck in terms of throughput and speed.

3.1.1 SD Card Insertion

However, what we have done so far is to calculate the value of each node belonging to a graph placed in the same file where we coded PageRank, resulting on a very slow data transfer rate through the **serial port** (9600 bits per second, corresponding to the

UART port). This is very inconvenient, not suitable for a modular implementation. That is the reason why graphs should be defined in a file that would enter a different physical port. There were several options attending to the possibilities of the FPGA, but after considering the capability of each input port, we decided to work with an SD card.

When using the SD card port as input, there is the problem that a baremetal FPGA (without any Operative System) cannot understand the binary content of the card on its own. This issue could be solved by using the *xilffs* library (*Generic Fat File System Library*), that builds a memory system by the usage of Xilinx proprietary commands that mount the SD card and settle a virtual disk where the graph file will be placed. This "driver" also turns the content of the textfile from ASCII characters to numbers.

The first step to implement the SD card into the project consists on declaring the SD-card device library and two huge buffers that will hold the origin and the destination nodes for all the graph edges. A pointer to the file and the name of the textfile declarations are also required.

Listing 3.1: SD-card input driver: libraries and global variables

```
#include "xsdps.h" // SD device library

/************************** Function Prototypes ******************************/
int FileInitialization(void);

/************************** Variable Definitions *****************************/
static FIL fil;    /* File object */
static FATFS fatfs;

/* To test logical drive 0, FileName should be "0:/<File name>" or "<file_name>". For
    logical drive 1, FileName should be "1:/<file_name>" */

static char FileName[32] = "FILE.txt"; // Name of the textfile (32 characters MAX)
static char *SD_File; // Pointer to the file

#ifdef __ICCARM__
#pragma data_alignment = 32
u8 DestinationAddress[10*1024]; // Buffer for the destination node of the links
#pragma data_alignment = 32
u8 SourceAddress[10*1024]; // Buffer for the origin node of the links
#else
u8 DestinationAddress[10*1024*1024] __attribute__ ((aligned(32)));
u8 SourceAddress[10*1024*1024] __attribute__ ((aligned(32)));
#endif
```

When the program is executed, the *main* function will call the *FileInitialization* function, in charge of managing the data from the SD-card.

Listing 3.2: SD-card input driver: main function calling

```
/*****************************************************************************/
int main(void) {
int Status;
...
Status = FileInitialization();
...
}
```

Once the execution reaches the *FileInitialization* function, a classical file system performance is emulated: the file system is mounted, then the textfile will be opened. The pointer will be required to indicate the beginning of file, after the file will be read, finally, the file will be closed.

Listing 3.3: SD-card input driver. FileInitialization function (1): filling the data buffers

```
/*****************************************************************************/
int FileInitialization(void) {
   FRESULT Res;
   UINT NumBytesRead;
   u32 FileSize = (1451*1024*1024);
   TCHAR *Path = "0:/"; // To test logical drive 0, Path should be "0:/". For logical
       drive 1, Path should be "1:/"

   Res = f_mount(&fatfs, Path, 0); // Register volume work area, initialize device
   if (Res != FR_OK) return XST_FAILURE;

   SD_File = (char *)FileName;
   Res = f_open(&fil, SD_File, FA_READ); // Open file
   if (Res) return XST_FAILURE;

   Res = f_lseek(&fil, 0); // Pointer to beginning of file
   if (Res) return XST_FAILURE;

    // Read data from file
    Res = f_read(&fil, (void*)DestinationAddress, FileSize, &NumBytesRead);
   if (Res) return XST_FAILURE;

   Res = f_close(&fil); // Close file
   if (Res) return XST_FAILURE;
```

Thereafter, once the two data buffers contain the whole data, the pointer will travel through both buffers with the aim of turning each detected character to a digit of the corresponding number. When a blank space is detected, the array *tempArray[]* (in charge of storing each digit of a number) will hold the complete enteger.

Listing 3.4: SD-card input driver. FileInitialization function (2): file characters reading

```
   int count = 0; // Number of digits per number iterator
   int tempArray[10]={0}; // Temporary char, its digits will form the definitive number
   int i=0; // Coefficient for number of element of DestinationAddress array

   // Getting number of nodes
   while(DestinationAddress[i]<58 && DestinationAddress[i]>47) {
      tempArray[count]= (int)DestinationAddress[i] - 48;
      count++;
      i++;
   }

   int n = sd_input(tempArray, count);
   count=0;
   i++;

   // Getting number of edges
   while(DestinationAddress[i]<58 && DestinationAddress[i]>47) {
      tempArray[count]= (int)DestinationAddress[i] - 48;
      count++;
      i++;
   }

   int e = sd_input(tempArray, count);
   count=0;
   i++;
   ...
}
```

After the array *tempArray[]* is full of digits (the whole number is stored), the *SD_input* function shows up, turning the array to the corresponding integer, thanks to *pow()* function, included in the *math.h* library that was declared at the top of the program. That is how all the read digits are joined, obtaining the corresponding number.

Listing 3.5: SD-card input driver: char to integer converter

```
1
2  /********************** SD card char to int converter **********************/
3
4   int sd_input(int tempArray[], int count) {
5      int number = 0;
6
7      for(int j=(count-1); j>=0; j--) {
8              number+=tempArray[count-j-1]*pow(10,j);
9              tempArray[count-j-1]=0; // reset
10     }
11     return number;
12  }
```

After applying these changes, programming the board and executing the program, we found out that using the SD card softly impacted on latency, due to the need of mounting a baremetal file system, leading to a slow increment in execution time.

Optimization	Average time ($\mu seconds$)
-O0	6295.70
-O1	4517.01
-O1 -fstrict-overflow -foptimize-sibling-calls -fpartial-inlining	4419.85
-O2	4405.12
-O2 -ftree-loop-vectorize	4348.47
-O3	4241.92

Table 3.1: Timing results (including SD-card input, without hardware or software modifications) over a small graph (450+ nodes, 2025+ edges)

Optimization	Average time ($\mu seconds$)
-O0	6298.49
-O1	4527.34
-O1 -fstrict-overflow -foptimize-sibling-calls -fpartial-inlining	4423.37
-O2	4413.28
-O2 -ftree-loop-vectorize	4358.21
-O3	4248.13

Table 3.2: Timing results (including SD-card input, with PageRank hardware block in the architecture) over a small graph (450+ nodes, 2025+ edges)

It can be observed that the final average time (*Table 3.1*) is quite worse than the one in the previous configuration (*Table 3.2*): the program accesses a textfile placed in a mounted file system, not directly in the code as in the baseline execution. The remaining time is lost in the typical SD-card file reading procedure.

Optimization	Speed up
-O0	1.000
-O1	1.002
-O1 -fstrict-overflow -foptimize-sibling-calls -fpartial-inlining	1.000
-O2	1.001
-O2 -ftree-loop-vectorize	1.002
-O3	1.001

Table 3.3: Speed-up (SD-card with no optimization / SD-card with optimization) over a small graph (450+ nodes, 2025+ edges)

A remarkable fact is that the speed-up results (*Table 3.3*) do not show important differences between the implementation with an specific PageRank hardware block (that includes HLS directives) and the baseline without an inserted hardware block. This leads to the conclusion that including a designed IP in a Vivado project is not very effective in our case, as well as it consumes more resources and does not give back any benefit. Later in this project, the PageRank hardware block will be deprecated and will not be used anymore.

3.1.2 Appropriate Memory Distribution

Timing results hardly improved after introducing the functionalities of the SD card, but circumstances change when dealing with memory organization. In *Xilinx SDK*, there is an specific file that gets along in this issue, specially the memory addresses each part of the program occupies and the stack and heap memories' measures, called *linker script*.

In order to configure this file, it is necessary to right click on the *SDK* project folder, choosing "*Generate linker script*".

Once a window shows up and clicked on *Advanced* tab (*Figure 3.1*), there are several options that can be modified, but three stand out: the **Code Section Assignments**, in charge of deciding which memory is most suitable for the *code* (instructions); The **Data Section Assignments**, managing the place where **data** stays, and the **Heap and Stack Section Assignments**, the most relevant option in terms of optimization; It makes the difference from the previous ones due to the fact that it allows to **assign an specific size for heap and stack memories**, appart from the memory part it occupies.

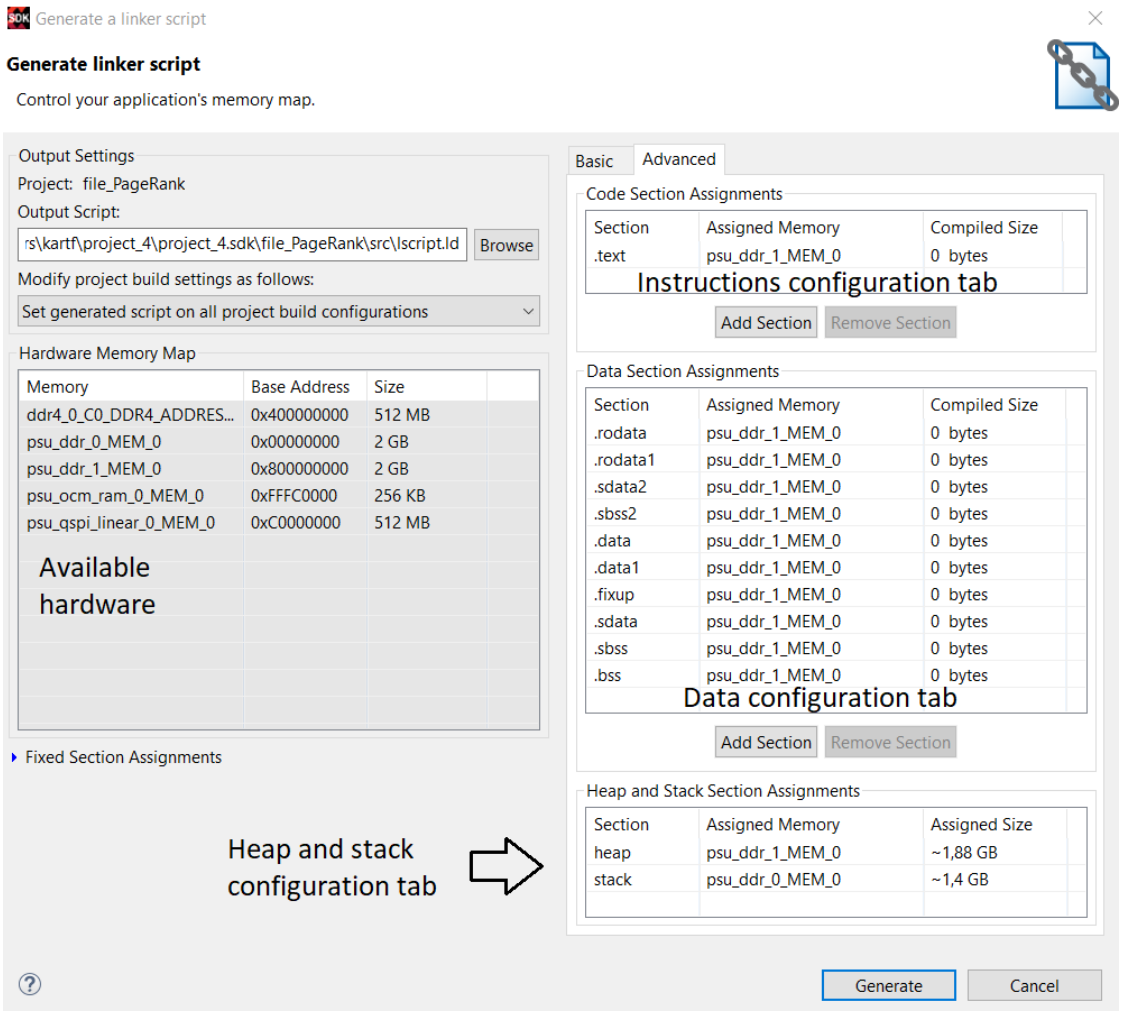


Figure 3.1: Linker script generation tab

After the memory features have been correctly adjusted, we will click *Generate* and changes will be applied to the project, so after programming the FPGA and executing the program again, performance improves in a very notorious way. Furthermore, when including new *compiler flags*, such as *-Ofast*, *-pipe* and *-funrol-loops*, average time reduces on 0.5 milliseconds, implying the best speed result so far (*Table 3.4*).

Optimization	Average time ($\mu seconds$)
-O0	4007.09
-O1	2343.92
-O1 -fstrict-overflow -foptimize-sibling-calls -fpartial-inlining	2242.29
-O2	2228.00
-O2 -ftree-loop-vectorize	2192.76
-O3	2044.80
-Ofast -pipe	1513.77
-funroll-loops	1512.74

Table 3.4: Timing results (with SD card input, pragmas, PageRank block, 252KB stack and 1.62GB heap and -mcmodel=large flag) over a small graph (450+ nodes, 2025+ edges)

3.1.3 Large Scale Implementation

Remapping memory is one of the best ways to reduce the execution time, leaving the path free to bigger graphs. After testing around twenty-five graphs containing big differences of quantity of nodules and links between them, the biggest graph that could be reached covered **more than 36000 elements and around 1.6 million edges, leading to an average time of almost 4 seconds** (*Table 3.5*).

Graph size	Average time ($\mu seconds$)
453 nodes, 2025 edges	1513.00
1446 nodes, 59589 edges	52888.56
3482 nodes, 155043 edges	159948.10
18448 nodes, 973918 edges	1902219.65
36364 nodes, 1590651 edges	3854160.64

Table 3.5: Large-scale graphs timing results (Optimized configuration, with 1'62GB heap in DDR and 252KB stack in OCM RAM)

In *Figure 3.2*, the x-scale describes the main features of the considered graph (number of nodes and links between them), and the y-scale represents the execution time.

It is remarkable the way the timing curve stabilizes below the fourth graph, meaning that when approaching the million edges the network turns more complex, increasing significantly the execution time. If the board, with the actual configuration, could hold a denser-edged input, timing would extend to quite higher numbers, as it can be seen that the curve starts to turn to an **exponential function** when the number of nodes and edges increase abruptly. Consequently, it makes intelligible that our FPGA cannot support them right now.

Nonetheless, the biggest reached graph would not be very useful in real life, so it is required to find a way of inserting larger inputs, which is described in the next sections.

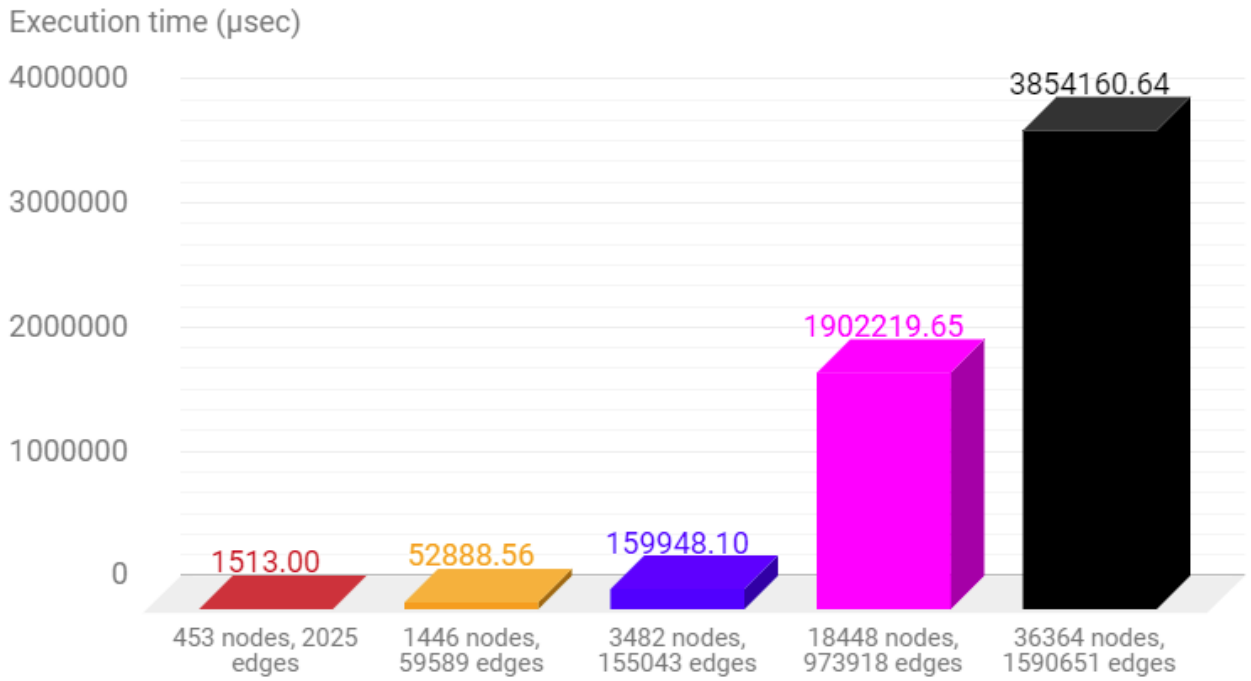


Figure 3.2: Large-scale graphs timing comparison (Optimized configuration, with 1'62GB heap in DDR and 252KB stack in OCM RAM)

3.1.4 Improving Hardware Design

This practical application of algorithm acceleration has been built so far on the idea of accomodating software which is programmed directly on the hardware of the FPGA. Nevertheless, hardware organization is one of the main factors that enforce the latency of the execution, and it is the issue that is going to be improved in this paragraph.

Instead of choosing an *Example Project* block design, we decided to build my own hardware organization from zero. There are several components which are essential for our purposes:

- **Zynq UltraScale+ MPSoC**: the processor is the most important part of the FPGA.
- **DDR4 RAM memory**: it is the fastest memory placed in the FPGA, and it should become the first option when storing or importing data is required.

Once these two IPs are positioned in the block design, we chose the autobuild option in order to set automatically all the connections between the components, without including the specific *PageRank* hardware block that did not bring back any remarkable benefit (*Figure 3.3*).

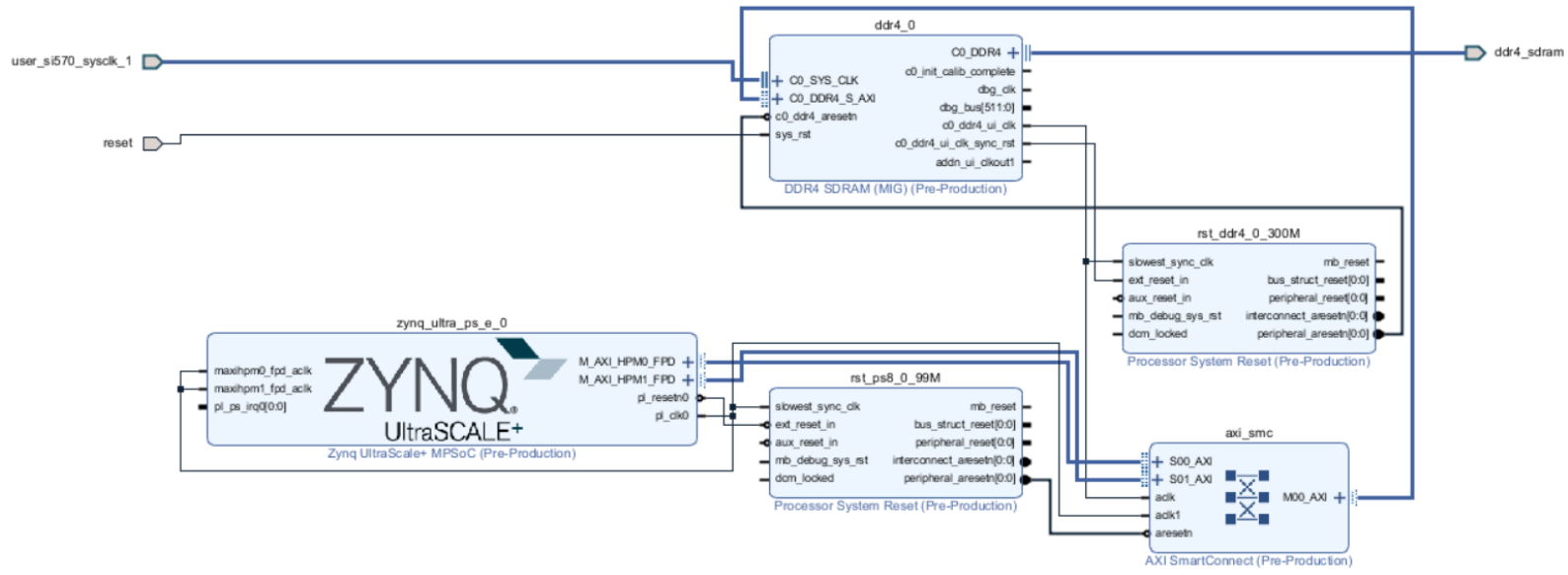


Figure 3.3: Block design of the optimized project in Vivado

Power and components usage reports (*Table 3.6* and *Table 3.7*) reflected a small upturn talking about on-chip power, but a big reduction of used logic gates. When tested on *Ubuntu 16.04 LTS*, it also brang good results (Appendix C).

On-chip power	4.773W	*Dynamic power*	4.041W (85%)	*Clocks*	0.160W
				Signals	0.086W
				Logic	0.159W
				BRAM	0.021W
				DSP	0.001W
				PLL	0.059W
				MMCM	0.114W
				I/O	0.247W
				PS	3.194W
		Static power	0.731W (15%)	*PL*	0.631W
				PS	0.100W

Table 3.6: Windows final implementation on-chip power

Element	*BRAM_18K*	*DSP48E*	*FF*	*LUT*	*URAM*	*WNS*	*WHS*	*WPWS*
Quantity	25.50	3	18658	16563	0	0.617ns	0.014ns	0.058ns

Table 3.7: Windows final implementation component usage and critical path stats

After the bitstream is generated, hardware has to be exported to *Xilinx SDK* and the previous explained ways of software optimizing are applied. The configuration that best suited the considered graphs was: **1.88GB heap on DDR0**, **1.4GB stack on DDR1**, and ***-Ofast -pipe -funroll-loops* compiler flags addition.**

After changes are saved, the project is programmed on the FPGA and the algorithm is executed. These features resulted on a impressive improvement on performance and the capability of elements and links, **allowing the FPGA to work on graphs with almost 300000 nodes and more than 2300000 edges** (*Table 3.8* and *Figure 3.4*, where the x-scale represents the quantity of nodes and links of each graph, and the y-scale shows the required time to execute the algorithm).

Graph size	Average time ($\mu seconds$)
36364 nodes, 1590651 edges	2033100.06
325729 nodes, 1497134 edges	3082364.29
281903 nodes, 2312497 edges	2901177.71

Table 3.8: Large-scale graphs timing results (new design, optimized configuration, with everything in DDR, 1'88GB heap and 1'4GB stack)

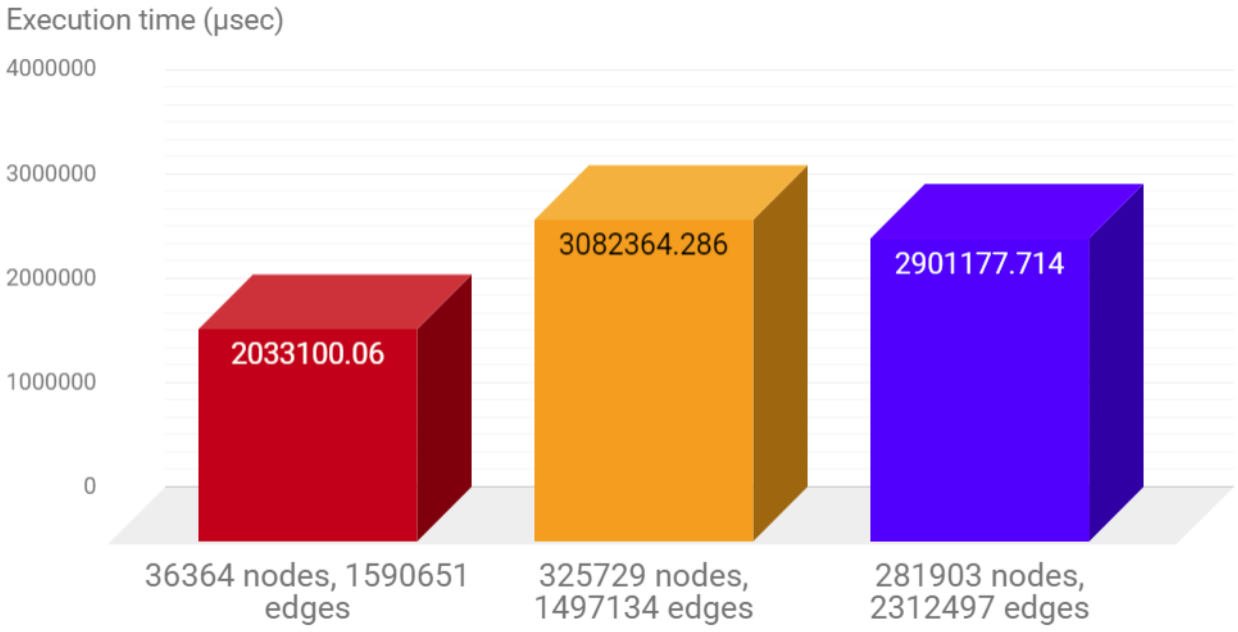


Figure 3.4: Large-scale graphs timing comparison (new design, optimized configuration, with everything in DDR, 1'88GB heap and 1'4GB stack)

In order to end with the monocore optimization, it was considered a **massive graph** aiming a customized configuration that enforced its performance.

Heap and stack memories configuration was modified, reaching **1.88GB heap and 11.64MB stack** (*Figure 3.5*), executing PageRank function in **2032818.84 μseconds** for a graph where 36364 nodes represent pages from *Texas University* and 1590651 directed edges represent hyperlinks between them.

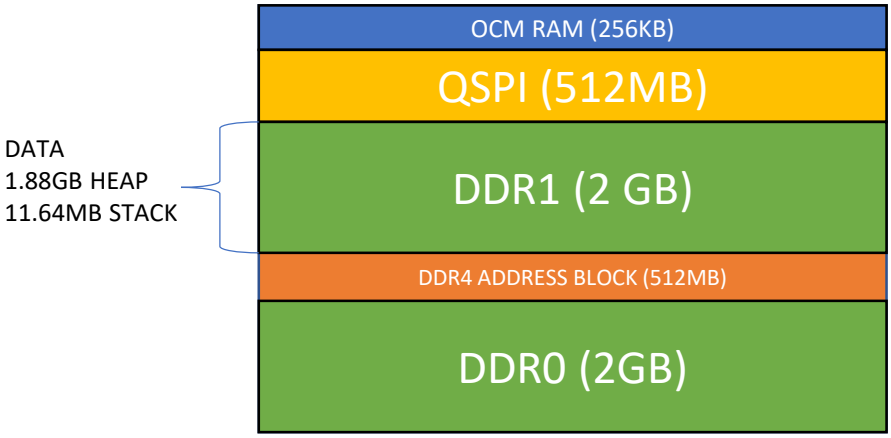


Figure 3.5: Final memory distribution

Finally, when using as input graph the biggest available one (281903 nodes and 2312497 edges), a network representing nodes and hyperlinks of the webpage of *Stanford University* (*stanford.edu*), its performance also improved, reaching **2900341.48 μseconds**.

3.2 Multicore Performance

After building a monocore PageRank on a baremetal implementation, testing the execution of the algorithm in several processors at the same time is an unavoidable issue.

In order to make it possible, SDK projects for each *ARM Core* need to be defined in the project settings before creating it. The programs should work normally in each processor. However, as we are executing the same code in all the projects, each code accesses the SD card and its respective file system at the same time, causing **collisions** (*Figure 3.6*). The SD-card input driver needs to be launched in different periods of time.

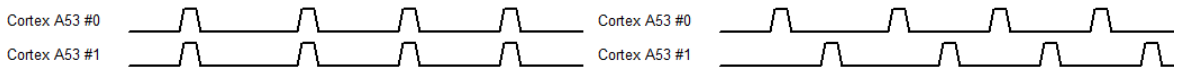


Figure 3.6: 2-Core file system accesses. Without manual delays (left), with manual delays (right)

If the project had been powered by an Operative System such as *Ubuntu* or *CentOS*, it would automatically take control of time distribution, delaying the processes that would be possible to run into each other. Nevertheless, this implementation is **baremetal**, and it exists a basic order that skips this problem manually: using big *for* loops that do not contain instructions.

Furthermore, code might not be executed by the FPGA due to memory requirements, specially **stack** and **heap**, so the *linker script* needed to be modified. We finally decided to spread the total size equally, giving approximately 1GB for each memory block (**972 MB**). This change resulted on a bottleneck that does not allow the engineer to implement the algorithm on more than two cores, that brought back the timing results detailed on *Table 3.9* and *Figure 3.7*, where it is compared the timing performance difference (y-scale) between the two working processors (x-scale).

Core	Average time ($\mu seconds$)
Cortex A53 #0	2899348.042
Cortex A53 #1	5736850.54

Table 3.9: Multicore timing results (972MB heap and 972MB stack)

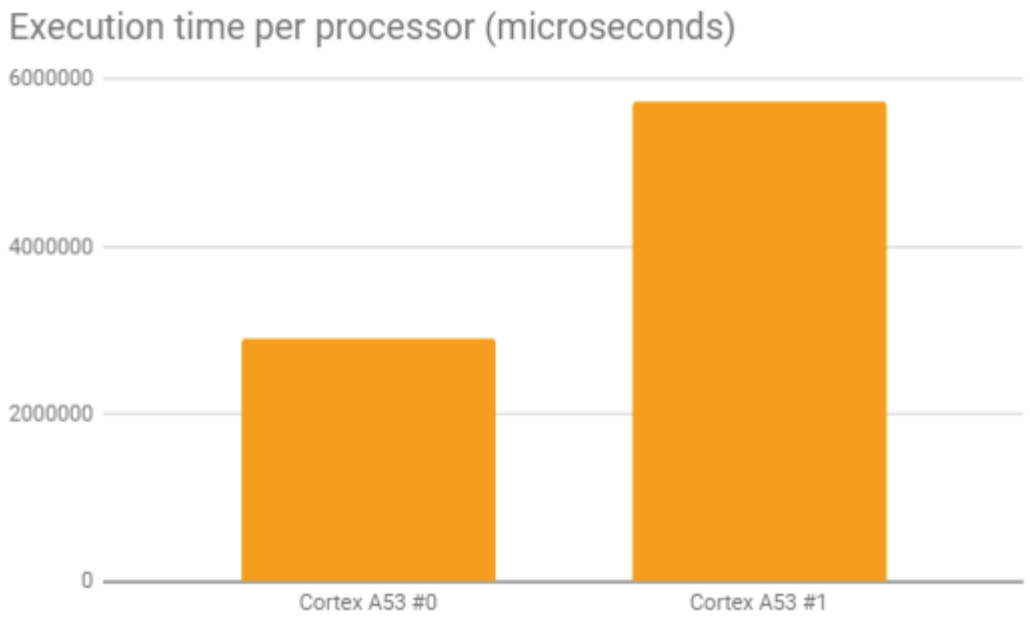


Figure 3.7: Multicore timing comparison (972MB heap and 972MB stack)

3.3 Conclusions of the Practical Application

After analyzing and executing different ways of improving the board's performance, we have reached several conclusions.

In the first sections, we put into practice an unoptimized algorithm, without inserting an specific PageRank hardware block into the reference design. However, when we added it, the speedup was less than one: **reserving a fixed quantity of resources in order to execute an algorithm faster is not always the best choice**. In our case, the FPGA works faster without including a hardware block into the design of Vivado, as it can be seen in *Figure 3.8*, where the red line represents the unoptimized implementation and the blue one shows how it affects the introduction of a PageRank IP into the design.

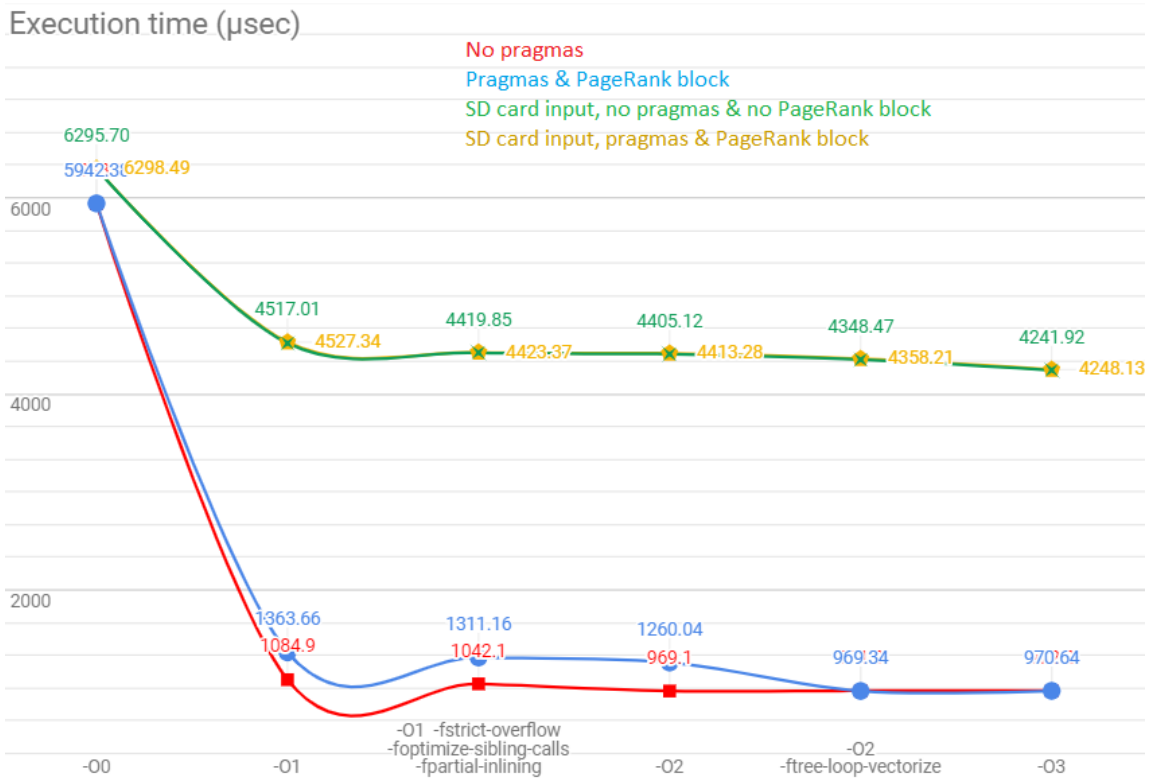


Figure 3.8: Baseline timing results comparison

The next step consisted on bringing to stage the SD card and its portability advantages. Implying a file system will always increase the execution time (green and yellow lines of *Figure 3.8*), as it is required to mount it, find the file, point to its beginning, read the data, and to close the file. Nonetheless, **using an SD card is more convenient**, due to the higher reading speed rates compared to other input ports. On the other hand, in *Figure 3.8* the difference between optimized and unoptimized paths is minimal, so we will not use *PageRank* IP block in order to let the FPGA choose the required resources in each execution. The baseline implementation has been set up.

Once the bedrock of the project is built, it was time to improve those results. The *linker script* is modified, altering the location of data, code and heap and stack memories. That is our way to avoid the problem of poor data locality and uncontrolled memory accesses. In fact, **moving data and instructions to other memory appropriate positions granted a reduction of execution time from 4248.13 μseconds to 1512.74 μseconds (speedup of 2.8)**. That relocation also opened the gate of real-life graphs, reaching more than 36000 nodes and almost 1.6 million links.

First software changes brought good results, but **there was an structural issue that had to be solved: the Vivado design**. A simpler design was tested, consuming less resources and enabling the entrance of even bigger graphs than before ones. **After optimizing software features on the new BSP, it could be possible to execute graphs of almost 300k nodes and 2.5 million edges, ten times more nodes and 50% more links.**

Chapter 4

Conclusions and Future Work

This end-of-degree project is focused on building an algorithm accelerator that increases the performance of the executions of an algorithm on an FPGA. For this purpose, it was used a laptop that can program a considered board, in our case a *ZCU102-ES2*.

We employed a basic program on the device, which was optimized with several options that reduced the execution time by rellocating data and instructions, modifying the size of the memories and using compiler flags. Finally, it resulted on higher speed rates, opening doors to real-life graphs that are run in a small quantity of time.

This work put the spotlight on issues from different fields treated along the degree:

- **Memory slowness**: the slow memory speed and data memory accesses without control issues are partially avoided thanks to modifying features of a *linker script*.
- **Compiler options are very relevant**: using a compiler that has not been configured and works by default is not desirable. When choosing the appropriate settings, performance can be improved easily.
- **Maths make computing simpler**: algorithms can be coded as their theorical description, but it usually exists an approximation that reduces substantially the time needed to execute it.
- **Proprietary solutions are not a good choice**: an *OpenCL* code could have solved the timing problem in a faster way, but it would not have been possible to use it in other different board or device. The exposed procedure of this work can be used in any FPGA, just deploying *Vivado HLx* software suite.

Despite the big quantity of advances reached in this project, there are several paths that could be followed, obtaining a richer work as a result.

4.1 OS insertion with PetaLinux

The multicore implementation that is proposed in this end-of-degree project is based on a **baremetal** point of view: the different components of the board are controlled manually by the engineer, which is not the most suitable option.

The distinct parts of the FPGA cannot be under the supervision of the expert constantly, and that issue can be solved by the introduction of an **Operative System** that distributes the access according to the requirements of each process, thanks to a tool based on **PetaLinux**. In this work, it was tested the insertion of *Ubuntu* and *CentOS*, but it was finally deprecated. However, it is the most logic way of improving performance.

4.2 Input Data through Ethernet

We decided to work with the SD-card port, but there is a better option but less portable: deploying Ethernet cable. Its throughput is quite higher, and data could be directly transferred from the host computer. On the other hand, it would be necessary to find the procedure to move that data from the computer to the board, as *Xilinx SDK* prioritizes the use of flashcards.

4.3 Finding the Available Exact Size of Graphs

The number of nodes and edges of the graphs used in the project are predefined by networks existent in real infrastructures, but it would be a good idea to generate our own graphs in order to test which is the approximate size that we could use as an input of *PageRank*. There is a synthetic graph generator that enables this option, called ***gpart***.

4.4 Graph Partitioning

Graph partitioning is a practice that can be very useful in terms of evaluating the capabilities of an FPGA when working with multicore implementations.

This issue can be treated in different ways: split up too big graphs so each core executes the algorithm with each fragment, or generating directly graphs of similar size, enough to fulfil the capacity of each processor of the board.

Appendix A

ARM Cortex Processors Family

ARM Cortex series [16] provide different cores depending on the functionality they are going to perform and their scalability. It can be distinguished three categories:

- **Cortex-A**: deployed for performance-intensive implementations.
- **Cortex-R**: high-performance cores specially designed for real-time applications.
- **Cortex-M**: used in a wide range of embedded systems.

Appendix B

PageRank Code

The code of the algorithm (using a pseudocode as reference [17]) aims to analyze the input graph stored in the SD card, and addresses its data in three buffers:

- **FileSize**: saves the whole graph data. From this buffer we will extract the different origin and destination nodes for each link, after stored in two new buffers.
- **DestinationAddress**: where all the corresponding destination nodes will be kept.
- **SourceAddress**: buffer where it will be stored the origin nodes for each edge or link.

When the three buffers hold all the information, then PageRank algorithm is computed and brings back its results for each node, including the time that it took to execute the complete algorithm (*Figure B.1*).

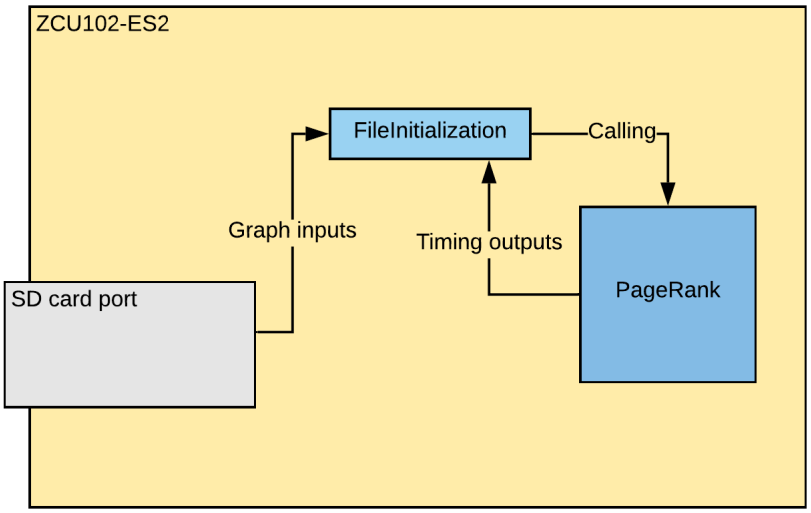


Figure B.1: PageRank code scheme

Listing B.1: Used libraries in PageRank code

```
/***************************** Include Files *********************************/

#include "xparameters.h" // SDK generated parameters
#include "xsdps.h"    // SD device driver
#include "xil_printf.h"
#include "ff.h"
#include "xil_cache.h"
#include "xplatform_info.h"
#include <stdlib.h>
#include <math.h> // Math library
#include "xtime_l.h" // XTime measure

/************************** Function Prototypes ******************************/
int FileInitialization(void);

/************************** Variable Definitions *****************************/
static FIL fil;    // File object
static FATFS fatfs;

/*
 * To test logical drive 0, FileName should be "0:/<File name>" or
 * "<file_name>". For logical drive 1, FileName should be "1:/<file_name>"
 */
static char FileName[32] = "FILE.txt";
static char *SD_File;
```

Listing B.2: SD-card graph number of nodes and edges reading

```
// Data buffer declaration
#pragma data_alignment = 32
u8 DestinationAddress[10*1024];
#pragma data_alignment = 32
u8 SourceAddress[10*1024];
TCHAR *Path = "0:/"; // Logical drive declaration
f_mount(&fatfs, Path, 0); // File system mounting
f_open(&fil, SD_File, FA_READ); // Open file with read permissions
f_lseek(&fil, 0); // Pointer to beginning of the file
f_read(&fil, (void*)Data_buffer, FileSize, &NumBytesRead); // Read data from file
f_close(&fil); // Close file

int count = 0; // Number of digits per number iterator
int tempArray[10]={0}; // Temporary char, after its digits will form the definitive
    number
```

```
XTime_GetTime(&tStart);
Turn each character to integer and join all digits to form the whole number of nodes
    and edges
```

Listing B.3: PageRank initialization

```

u32 FileSize = (1451*1024);
int granularity = 1;

float *val = calloc(n, sizeof(float));
int *col_ind = calloc(n, sizeof(int));
int *row_ptr = calloc(n+1, sizeof(int));

// The first row always starts at position 0
             row_ptr[0] = 0;

             int cur_row = 0;
             int iterator = 0; // Number of analyzed edges iterator
             int j = 0;
             // Elements for row
             int elrow = 0;
             // Cumulative numbers of elements
             int curel = 0;

while(iterator < Number of edges) {
    Read origin node
    Read destination node
     if (Origin node > cur_row) { // change the row
               curel = curel + elrow;
               for (int k = cur_row + 1; k <= Origin node; k++) row_ptr[k] = curel;
               elrow = 0;
               cur_row = Origin node;
    }

    val[iterator] = 1.0;
    col_ind[iterator] = Destination node;
    elrow++;
    iterator++;
}
     row_ptr[cur_row+1] = curel + elrow - 1;

     // Fix the stochastization
    int out_link[n];
```

```
    for(i=0; i<n; i++) {
            out_link[i] = 0;
    }

    int rowel = 0;
    for(i=0; i<n; i++){
       if (row_ptr[i+1] != 0) {
             rowel = row_ptr[i+1] - row_ptr[i];
             out_link[i] = rowel;
       }
     }

    int curcol = 0;
    for(i=0; i<n; i++) {
      rowel = row_ptr[i+1] - row_ptr[i];
      for (j=0; j<rowel; j++) {
       val[curcol] = val[curcol] / out_link[i];
       curcol++;
      }
    }

    /* PageRank algorithm */
    PageRank(n, val, col_ind, row_ptr, granularity);
    XTime_GetTime(&tEnd);
    xil_printf("%d\n", 2*(tEnd - tStart)); // Zynq SoC's counter increases every two
        clock cycles

return 0;

}
```

Listing B.4: PageRank algorithm code

```

int PageRank(int n, float val[], int col_ind[], int row_ptr[], int granularity) {

     float d = 0.85; // Set the damping factor 'd'
    float p[n]; // Initialize p[] vector
    for(int i=0; i<n; i++) p[i] = 1/n; // P_INITIALIZE_1DIVN:
    float p_new[n]; // Initialize new p vector
    int looping = 1; // Set the looping condition

    // LOOPING_WHILE:
      while (looping){
```

```
12          // Initialize p_new as a vector of n 0.0 cells
13          // P_INITIALIZE_0:
14             for(int i=0; i<n; i++) p_new[i] = 0.0;
15
16             int rowel = 0;
17             int curcol = 0;
18
19             // Page rank modified algorithm + parallelization
20             // PAGERANK_LOOP1:
21             // This part of the code uses many resources
22                for(int i=0; i<n; i = i + granularity){
23                    rowel = row_ptr[i+1] - row_ptr[i];
24                    // PAGERANK_SUBLOOP:
25                   for (int j=0; j<rowel; j++) {
                         p_new[col_ind[curcol]]=p_new[col_ind[curcol]]+val[curcol]*p[i];
26                    curcol++;
27                    }
28              }
29
30           // PAGERANK_LOOP2:
31              for(int i=0; i<n; i++) p_new[i] = d * p_new[i] + ((1.0 - d) / n);
32
33            // TERMINATION: check if we have to stop
34            double error = 0.0;
35            // ERROR_CHECK:
36            for(int i=0; i<n; i++) error = error + fabs(p_new[i] - p[i]);
37            // If two consecutive instances of pagerank vector are almost identical,
                  stop
38            if (error < 0.00000000000000000000000000000000000000000000000000000001)
                  looping = 0;
39
40            // Update p[]
41            // P_UPDATE:
42               for (int i=0; i<n;i++) p[i] = p_new[i];
43
44            // Increase the number of iterations
45            k = k + 1;
46      }
47      return 0;
48  }
```

Appendix C

Ubuntu Optimized Block Design

A deprecated path of investigation was based on building the project with *Ubuntu 16.04 LTS* as the Operative System used in the laptop where *Vivado HLx* was installed. If the optimized block design of Windows is implemented on *Linux*, it resulted on better on-chip power and components usage (*Table C.1* and *Table C.2*).

On-chip power	4.724W	*Dynamic power*	3.993W (85%)	*Clocks*	0.162W
				Signals	0.078W
				Logic	0.121W
				BRAM	0.022W
				DSP	0.001W
				PLL	0.059W
				MMCM	0.114W
				I/O	0.245W
				PS	3.190W
		Static power	0.731W (15%)	*PL*	0.631W
				PS	0.100W

Table C.1: Ubuntu final implementation on-chip power

Element	*BRAM_18K*	*DSP48E*	*FF*	*LUT*	*URAM*	*WNS*	*WHS*	*WPWS*
Quantity	0	3	25463	23719	0	0.736ns	0.016ns	0.058ns

Table C.2: Ubuntu final implementation component usage and critical path stats

List of Figures

List of Tables

Listings

Bibliography

[1] Xilinx, "Vivado Design Suite User Guide: Design Analysis and Closure Techniques". UG906, 2012.

[2] Úrsula Iturrarán-Viveros and Miguel Molero-Armenta, "GPU computing with OpenCL to model 2D elastic wave propagation: exploring memory usage.", in *Computational Science & Discovery, Vol. 8*, 2015.

[3] Xilinx, "ZCU102 Evaluation Board User Guide". UG1182, 2019.

[4] Xilinx. "Board Support Packages", `https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/SDK_doc/concepts/sdk_c_bsp.htm`

[5] Brahim Betkaoui, David B. Thomas, Wayne Luk and Natasa Przulj, "A Framework for FPGA Acceleration of Large Graph Problems: Graphlet Counting Case Study", in *2011 International Conference on Field-Programmable Technology*, 2011.

[6] Xinyu Chen, Ronak Bajaj, Yao Chen, Jiong He, Bingsheng He, Weng-Fai Wong and Deming Chen, "On-The-Fly Parallel Data Shuffling for Graph Processing on OpenCL-based FPGAs", *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019.

[7] Bo Joel Svensson and Rakesh Tripathi, "Getting Started with OpenCL on the ZYNQ", `https://svenssonjoel.github.io/writing/ZynqOpenCL.pdf`, 2018.

[8] Maciej Besta, Dimitri Stanojevic, Johannes de Fine Licht, Tal Ben-Nun and Torsten Hoefler, "Graph Processing on FPGAs: Taxonomy, Survey, Challenges. Towards Understanding of Modern Graph Processing, Storage, and Analytics.", *arXiv preprint arXiv:1903.06697v3*, 2019.

[9] Yaman Umuroglu and Magnus Jahre, "Hybrid breadth-first search on a single-chip FPGA-CPU heterogeneous platform", in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, 2015.

[10] Soroosh Khoram, Jialiang Zhang, Maxwell Strange and Jing Li, "Accelerating Graph Analytics by Co-Optimizing Storage and Access on an FPGA-HMC Platform", in *2018 ACM/SIGDA International Symposium*, 2018.

[11] Free Software Foundation, Inc., "Using the GNU Compiler Collection (GCC)".

[12] Xilinx, "SDSoC Profiling and Optimization Guide". UG1235, 2016.

[13] Xilinx, "Improving Performance. Vivado HLS 2013.3 Version". 2013.

[14] Xilinx, "Xilinx Software Development Kit (SDK) User Guide - Getting Started with Xilinx SDK".

[15] Kartik Lakhotia and Viktor Prasanna, "Accelerating PageRank using Partition-Centric Processing", in *2018 USENIX Annual Technical Conference (USENIX ATC '18)*, 2018.

[16] Silicon Labs, "Which ARM Cortex Core Is Right for Your Application: A, R or M?".

[17] Shijie Zhou, Charalampos Chelmis and Viktor K. Prasanna, "Optimizing Memory Performance for FPGA Implementation of PageRank", in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2015.

[18] Scott Beamer, Krste Asanović, David Patterson, "Reducing Pagerank Communication via Propagation Blocking", in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017.

[19] Frank McSherry, "A uniform approach to accelerated PageRank computation", in *WWW '05: Proceedings of the 14th international conference on World Wide Web*, 2005.

[20] Xilinx, "SDSoC Environment User Guide". UG1027, 2018.

[21] Xilinx, "SDSoC Environment Tutorial". UG1028, 2017.

[22] Xilinx, "Vivado Design Suite User Guide". UG906, 2012.

[23] Xilinx, "SDAccel Environment User Guide". UG1023, 2019.

[24] Xilinx, "SDAccel Programmers Guide". UG1277, 2018.

[25] Jorge Luiz e Silva, Bruno de Abreu Silva, Joelmir Jose Lopes and Antonio Carlos F. da Silva, "Accelerating Algorithms using a Dataflow Graph in a Reconfigurable System", *arXiv preprint arXiv:1110.3655v1*, 2018.

[26] Takaaki Miyajima, David Thomas and Hideharu Amano, "An Automatic Mixed Software Hardware Pipeline Builder for CPU-FPGA Platforms", *arXiv preprint arXiv:1408.4969v1*, 2014.