

UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

**A hardware/software codesign for the chemical reactivity of
BRAMS**

Carlos Alberto Oliveira de Souza Junior

Dissertação de Mestrado do Programa de Pós-Graduação em Ciências
de Computação e Matemática Computacional (PPG-C²MC)

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Carlos Alberto Oliveira de Souza Junior

A hardware/software codesign for the chemical reactivity of BRAMS

Master dissertation submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP, in partial fulfillment of the requirements for the degree of the Master Program in Computer Science and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Eduardo Marques

USP – São Carlos
August 2017

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

S684h Souza Junior, Carlos Alberto Oliveira de
A hardware/software codesign for the chemical
reactivity of BRAMS / Carlos Alberto Oliveira de
Souza Junior; orientador Eduardo Marques. -- São
Carlos, 2017.
109 p.

Dissertação (Mestrado - Programa de Pós-Graduação
em Ciências de Computação e Matemática
Computacional) -- Instituto de Ciências Matemáticas
e de Computação, Universidade de São Paulo, 2017.

1. Hardware. 2. FPGA. 3. OpenCL. 4. Codesign. 5.
Heterogeneous-computing. I. Marques, Eduardo,
orient. II. Título.

Carlos Alberto Oliveira de Souza Junior

Um coprojeto de hardware/software para a reatividade
química do BRAMS

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Eduardo Marques

USP – São Carlos
Agosto de 2017

ACKNOWLEDGEMENTS

Firstly, I thank God for being able to fulfill a dream, for giving me health and introducing me wonderful people who supported me to walk through this path. One of them is my advisor, Prof. Eduardo Marques, who is not only wise and patient but also a great friend who has always been available to help me to overcome the obstacles in the way.

My sincere gratitude to all my undergraduate professors, in special, I thank Professors Evanise Caldas, Fábio Hernandez, and Mauro Mulati. My friends Lucas Lorenzetti and Paulo Urio for sharing the dull and funny moments. To my new friends from LCR (Erinaldo, Marcilyanne, and Rafael), who were very welcoming to me. I would also like to thank CNPq for their financial support.

Finally, I am deeply thankful to my family. In particular my parents, my grandparents and my sister Kelly, even with the distance they were always close to me in my thoughts. I love you all, and without your support, this dream would not come true.

“I love deadlines. I like the whooshing sound they make as they fly by.”

Douglas Adams

ABSTRACT

OLIVEIRA DE SOUZA JUNIOR, C. A. **A hardware/software codesign for the chemical reactivity of BRAMS**. 2017. 109 f. Master dissertation (Master student Program in Computer Science and Computational Mathematics) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

Several critical human activities depend on the weather forecasting. Some of them are transportation, health, work, safety, and agriculture. Such activities require computational solutions for weather forecasting through numerical models. These numerical models must be accurate and allow the computers to process them quickly. In this project, we aim at migrating a small part of the software of the weather forecasting model of Brazil, BRAMS — Brazilian developments on the Regional Atmospheric Modelling System — to a heterogeneous system composed of Xeon (Intel) processors coupled to a reprogrammable circuit (FPGA) via PCIe bus. According to the studies in the literature, the chemical equation from the mass continuity equation is the most computationally demanding part. This term calculates several linear systems $Ax = b$. Thus, we implemented such equations in hardware and provided a portable and highly parallel design in OpenCL language. The OpenCL framework also allowed us to couple our circuit to BRAMS legacy code in Fortran90. Although the development tools present several problems, the designed solution has shown to be viable with the exploration of parallel techniques. However, the performance was below of what we expected.

Keywords: Hardware, FPGA, OpenCL, codesign, heterogeneous-computing.

RESUMO

OLIVEIRA DE SOUZA JUNIOR, C. A. **A hardware/software codesign for the chemical reactivity of BRAMS**. 2017. 109 f. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

Várias atividades humanas dependem da previsão do tempo. Algumas delas são transporte, saúde, trabalho, segurança e agricultura. Tais atividades exigem soluções computacionais para previsão do tempo através de modelos numéricos. Estes modelos numéricos devem ser precisos e ágeis para serem processados no computador. Este projeto visa portar uma pequena parte do software do modelo de previsão de tempo do Brasil, o BRAMS — *Brazilian developments on the Regional Atmospheric Modelling System* — para uma arquitetura heterogênea composta por processadores Xeon (Intel) acoplados a um circuito reprogramável em FPGA via barramento PCIe. De acordo com os estudos, o termo da química da equação de continuidade da massa é o termo mais caro computacionalmente. Este termo calcula várias equações lineares do tipo $Ax = b$. Deste modo, este trabalho implementou estas equações em hardware, provendo um código portátil e paralelo na linguagem OpenCL. O framework OpenCL também nos permitiu acoplar o código legado do BRAMS em Fortran90 junto com o hardware desenvolvido. Embora as ferramentas de desenvolvimento tenham apresentado vários problemas, a solução implementada mostrou-se viável com a exploração de técnicas de paralelismo. Entretanto sua performance ficou muito aquém do desejado.

Palavras-chave: Hardware, FPGA, OpenCL, coprojeto, computação-heterogênea.

LIST OF FIGURES

Figure 1 – Simulation of CCATT-BRAMS system, figure from (LONGO <i>et al.</i> , 2013).	31
Figure 2 – Rosenbrock Method.	33
Figure 3 – Jacobi Method algorithm.	37
Figure 4 – Clock rate and power increase of eight generations of Intel microprocessors, figure from (PATTERSON; HENNESSY, 2012).	38
Figure 5 – OpenCL Data Structures – Consider Program as a single data structure; we replicated it to make the understanding easier.	40
Figure 6 – An example of how the global IDs, local IDs, and work-group indices are related for a two-dimensional NDRange. For this figure, we have the following indices: the shaded block has a global ID of $(g_x, g_y) = (6, 5)$, a work-group ID of $(w_x, w_y) = (1, 1)$ plus a local ID of $(l_x, l_y) = (2, 1)$, figure from (MUNSHI, 2009).	41
Figure 7 – Components from OpenCL system on Intel FPGAs, figure from (ALTERA, 2013).	42
Figure 8 – Partitioning of the FPGA. PCIe, DDR3 controller and IPs are every project of OpenCL, so only the remaining is available for the kernels, figure granted by André Perina.	42
Figure 9 – Implementation of local memory with three M20K blocks, figure from (INTEL, 2016a).	43
Figure 10 – Design flow with OpenCL, figure from (CZAJKOWSKI <i>et al.</i> , 2012b).	44
Figure 11 – Xilinx field-programmable gate array (FPGA) progression. (Price and power are per logic cell.), figure from (AHMAD <i>et al.</i> , 2016).	45
Figure 12 – Top-Level Component Implementation Block Diagram, figure from (BITTWARE, 2015).	46
Figure 13 – Driving factors in hardware/software codesign, figure from (SCHAUMONT, 2012).	48
Figure 14 – Speedup compared to CPU versions. The x dimension stands for matrix size, and y dimension speedup in FPGA.	50
Figure 15 – Generic representation of BRAMS system with a single process. In this figure, we present BRAMS over the South America with a single grid, yellow square represents Sparse1.3 running for all the points over the grid.	56

Figure 16 – Generic representation of BRAMS system with MPI processes. In this figure, we present BRAMS over the South America with a single grid distributed over N processes, each process executes Sparse 1.3 for its set of points of the grid. The shaded areas are the ghost zones, i.e. the shared data area.	56
Figure 17 – Generic representation of BRAMS coupled to the Jacobi method in Hardware. In this figure, we present BRAMS over the South America with a single grid, red square represents Jacobi hardware circuit. Such circuit computes all the points over the grid.	58
Figure 18 – Schematic of Jacobi method with multiple threads with dense representation.	59
Figure 19 – Each color represents one work-group, each work-group has 47 work-items. According to the verilog, two work-groups are executing at the same time.	60
Figure 20 – Interoperability of BRAMS and OpenCL. Fortran calls a function from the C host, which in turn is responsible to manage the device.	61
Figure 21 – Schematic of Jacobi method with multiple threads with sparse representation.	63
Figure 22 – Each color represents one work-group, each work-group has one work-item. In this manner, the number of work-items is equal to the number of work-groups.	64
Figure 23 – Schematic of Sparse Jacobi method with single thread.	66
Figure 24 – How pageable and pinned memory data transfer work, based on Harris (2012).	67
Figure 25 – Call Graph for BRAMS with chemical module disabled.	70
Figure 26 – Call Graph for BRAMS with chemical module enabled.	71
Figure 27 – Each MPI process accesses a copy of the kernel (an FPGA circuit).	72
Figure 28 – OpenCL data structures – Program, Device and Memory buffers are shared among MPI processes.	73
Figure 29 – Pipeline report for Jacobi multi-threaded dense.	74
Figure 30 – CPU communicates with FPGA for every iteration. CPU sends to the FPGA the initial data, after FPGA processing it, the FPGA returns the result to the CPU, which in turn computes the vector norm and decides if it sends another data or computes another iteration.	76
Figure 31 – Communication and execution time with Intel FPGA SDK profiling for kernels. Note that there is much more communication than computation.	77
Figure 32 – Efficiency of Jacobi multi-threaded dense with Intel FPGA SDK profiling for kernels. The red line points that the global memory reads (line 22) are the bottleneck of the application.	77
Figure 33 – Statistics of Jacobi multi-threaded dense with Intel FPGA SDK profiling for kernels.	77
Figure 34 – Efficiency of Jacobi multi-threaded sparse with Intel FPGA SDK profiling for kernels. Sparse format causes a severe drop of performance when saving the results back to the global memory.	78

Figure 35 – Execution and memory transfer time of Jacobi multi-threaded sparse with Intel FPGA SDK profiling for kernels. Note that transfer time did not improve due to variable sparsity of the matrices.	79
Figure 36 – Statistics of Jacobi multi-threaded sparse with Intel FPGA SDK profiling for kernels.	79
Figure 37 – Pipeline report for Jacobi single-threaded sparse.	81
Figure 38 – Optimum pipeline with II of 1, figure from Intel (2016a).	82
Figure 39 – Matrix-vector pipeline with II of 11, based on Intel (2016a).	82
Figure 40 – Pipeline report for Jacobi single-threaded sparse optimized.	83
Figure 41 – Efficiency for Jacobi single-threaded sparse. After the modifications, all the pipelines shows 100% of efficiency and almost zero stall.	84
Figure 42 – Execution and memory transfers time for Jacobi single-threaded sparse. Each bar in spjacobi_method1 means a complete execution over a matrix.	85
Figure 43 – Statistics for Jacobi single-threaded sparse.	85

LIST OF TABLES

Table 1 – Results from CLOC.	27
Table 2 – S5PH-Q Features	47
Table 4 – Table of comparison among related works	53
Table 6 – Results from Arch 1.	77
Table 7 – Timing results from Arch 1.	78
Table 8 – Results from Arch 2.	80
Table 9 – Timing results from Arch 2.	80
Table 10 – Results from Arch 3.	86
Table 11 – Tmining results from Arch 3.	86
Table 12 – Timing results from Arch 4.	87
Table 13 – Table of comparison among implementations.	87
Table 15 – Comparison among architectures	88

ACRONYMS

AOCL Altera OpenCL.

API Application Programming Interface.

ARM Advanced RISC Machines.

ATMET ATmospheric, Meteorological, and Environmental Technologies.

BRAMS Brazilian developments on the Regional Atmospheric Modelling System.

CB Carbon Bond.

CBEA Cell Broadband Engine Architecture.

CCATT Coupled Chemistry Aerosol–Tracer Transport.

CDFG Control-Data Flow Graph.

CPTEC Center for Weather Forecasts and Climate Studies.

CPU Central Processing Unit.

CSR Compressed Row Storage.

FINEP Financier of Studies and Projects.

FPGA Field-Programmable Gate Array.

GPU Graphics Processing Unit.

HDF5 Hierarchical Data Format.

HDL Hardware Description Language.

HIPA^{cc} Heterogeneous Image Processing Acceleration Framework.

IAG Institute of Astronomy, Geophysics and Atmospheric Sciences.

II Initiation Interval.

IME Institute of Mathematics and Statistics.

INPE National Institute for Space Research.

IP Intellectual Property.

IR Intermediate Representation.

JULES Joint UK Land Environment Simulator.

KPP Kinetic PreProcessor.

LCR Reconfigurable Computing Laboratory.

LE Logic Elements.

LU Lower Upper.

LUT Look-Up Table.

MPI Message Passing Interface.

MPICH Message Passing Interface CHameleon.

MRA Multiresolution Analysis.

NDRange N-Dimensional Range.

NetCDF Network Common Data Form.

OpenCL Open Computing Language.

PBL planetary Boundary Layer.

PCIe Peripheral Component Interconnect Express.

PDE Partial Differential Equations.

RACM Regional Atmospheric Chemistry Mechanism.

RAMS Regional Atmospheric Modeling System.

RELACS Regional Lumped Atmospheric Chemical Scheme.

RTL Register-Transfer Level.

SDK Software Development Kit.

SIMD Single Instruction, Multiple Data.

SOC System-On-Chip.

SOPC System-On-a-Programmable-Chip.

SpMV Sparse Matrix-Vector multiplication.

SRAM Static Random Access Memory.

SVD Singular Value Decomposition.

USP University of São Paulo.

CONTENTS

Acronyms	19
1 INTRODUCTION	25
1.1 Objective	27
1.2 Motivation	27
1.3 Document organization	28
2 FUNDAMENTAL CONCEPTS	29
2.1 Brazilian developments on the Regional Atmospheric Modelling System – BRAMS	29
2.1.1 <i>CCATT-BRAMS</i>	30
2.1.2 <i>Libraries</i>	32
2.1.3 <i>NetCDF</i>	32
2.1.4 <i>HDF5</i>	34
2.1.5 <i>Zlib</i>	34
2.1.6 <i>Szip</i>	34
2.1.7 <i>Mpich</i>	34
2.1.7.1 <i>MPI</i>	35
2.2 Linear Equation	35
2.3 Linear Solver	36
2.3.1 <i>Direct Method - LU</i>	36
2.3.2 <i>Iterative Method - Jacobi</i>	36
2.4 OpenCL	37
2.4.1 <i>Data structures for OpenCL</i>	38
2.4.2 <i>Data Parallelism</i>	39
2.4.3 <i>Task Parallelism</i>	40
2.5 Intel FPGA SDK for OpenCL	41
2.6 FPGA	44
2.7 Development Environment for OpenCL	45
2.7.1 <i>BittWare Board S5PH-Q</i>	46
2.8 Hardware/Software Codesign	46
2.9 Related Work	48

3	DEVELOPMENT OF THE CODESIGN FOR THE CHEMICAL RE- ACTIVITY OF BRAMS	55
3.1	CCATT–BRAMS software	55
3.1.1	<i>Interoperability with Sparse1.3a</i>	57
3.2	CCATT–BRAMS Codesign	57
3.2.1	<i>Jacobi Multi-threaded Dense</i>	57
3.2.2	<i>Interoperability with Jacobi Multi-Threaded Dense</i>	60
3.2.3	<i>Jacobi Multi-threaded Sparse</i>	62
3.2.4	<i>Interoperability with Jacobi Multi-Threaded Sparse</i>	63
3.2.5	<i>Jacobi Single-threaded Sparse</i>	64
3.2.6	<i>Interoperability with Jacobi Single-threaded Sparse</i>	67
4	RESULTS	69
4.1	Result analysis	69
4.2	Experiments	70
4.3	Results from Jacobi Multi-threaded Dense	73
4.4	Results from Jacobi Multi-threaded Sparse	78
4.5	Results from Jacobi Single-threaded Sparse	80
4.6	Results from Jacobi Single-threaded Dense	86
4.7	Results from Sparse1.3a	87
5	CONCLUSION	89
5.1	Limitations	89
5.2	Future Work	90
	BIBLIOGRAPHY	91
	APPENDIX	101
	APPENDIX	103
APPENDIX A	INSTALLATION	103
APPENDIX B	OBSERVING THE RESULTS	107
ANNEX A	USEFUL LINKS	109

INTRODUCTION

Weather forecasting is the utilization of science and technology to predict the state of the atmosphere at a provided location. The predictions require quantitative data from the current situation of the atmosphere at a given place, and scientific understanding of atmospheric processes to predict how the atmosphere will change ([HENKEL, 2015](#)).

According to [Society \(2015\)](#) the meteorological profession is responsible for two crucial services: weather forecasts and warnings. The government and industry use forecasts to protect life and property and to improve the efficiency of operations.

Several critical human activities depend on the weather forecasting. Some of them are transportation, health, work, and safety. Imagine an air travel where passengers do not know what are the risks ahead; a forest fire where firefighters have no clue where the fire will move ([LABORATORY, 2015](#)).

Forecasts based on temperature and precipitation are important to agriculture and so the traders of commodity markets. Weather forecasting is also critical to estimate the crop-disease spread ([WARNER, 2010](#)). Trying to predict the weather is not a new science.

Ancient civilizations started to forecast the weather at the very early ages. They used astronomical and meteorological events to keep track of seasonal changes in the weather. By the year 650 B.C., Babylonians tried to predict weather (short-term) through clouds patterns and optical phenomena, such as halos ([GRAHAM; PARKINSON; CHAHINE, 2002](#)).

By the end of Renaissance period, it had become clear to philosophers that forecasting based only on observations and assumptions was not an adequate method. They needed to improve their understanding of the atmosphere — the development of new tools was necessary to measure properties of the atmosphere, such as moisture, temperature, and pressure. The first tool dates back to 14th century, the hygrometer.

On the 16th century, Galileo Galilei invented the thermometer. Later in the 17th century,

Evangelista Torricelli created the barometer. Other tools came later on recently centuries (for example radiosonde and weather satellite). These instruments allowed us to create weather observation stations and the dissemination of them around the globe. Besides the tools, it was also necessary a better understanding of the atmosphere.

In the 19th century, the development of thermodynamics allowed meteorologists to set the fundamental physical principles that govern the flow of the atmosphere (LYNCH, 2008). In 1890, Cleveland Abbe acknowledged that meteorology is the application of hydrodynamics and thermodynamics to the atmosphere (WILLIS; HOOKE, 2006).

In 1904, Vilhelm Bjerknes published a paper in german (The Problem of Weather Forecasting from the Standpoint of Mechanics and Physics). In this paper, he introduces the hydro- and thermodynamics into meteorology. Vilhelm included the second law of thermodynamics in his set of equations; this error was corrected by Lewis Fry Richardson (GRØNØS, 2005). The latter made an estimate method for solving numerical equations — According to Richardson, it would be required 64 thousand people to predict the weather in time. It is clear that predicting the weather was impossible before computational era (LYNCH, 2008).

Predicting the weather became possible in the 20th century, with Von Neumann's ENIAC. Charney realized that could overcome Richardson's methods impracticability with new computers, and a revised set of equations where scientists could solve complex equations through numerical methods. In April 1950, Charney's research group succeed to predict the weather for 24 hours in the North America.

With the computer availability increasing in the 20th century, universities started offering courses on atmospheric modeling. At the time, people shared their time as both modelers and developers; the generated code had many errors, so it was necessary more human effort to correct it. The development in the field allowed the modelers to have available, for free, well-tested community, global and limited-area models, and access to full documentation, regular tutorials, and technical support (WARNER, 2010).

Last century was responsible for many scientific advances that allowed us to predict the weather phenomena. This prediction uses numerical models; we can classify such models according to their domain operation: Global (the entire Earth planet) and regional (e.g. Country, State, and City). The global models cannot represent accurately the regional weather phenomena due to limited computing power. On the other hand, regional models are more accurate (OSTHOFF *et al.*, 2012; LABORATORY, 2015). In this project, we are using [Brazilian developments on the Regional Atmospheric Modelling System \(BRAMS\)](#)¹, a Brazilian regional model.

Currently, BRAMS is the largest high-performance application in Brazil according to [Panetta \(2015\)](#). We used a PERL script to count the lines of BRAMS, with CLOC² we discovered

¹ <http://brams.cptec.inpe.br/>

² <http://cloc.sourceforge.net/>

that BRAMS has over 400 thousands lines of code in Fortran. We can see the results in Table 1. Since the application is huge, we used some tools available on the Internet to detect bottlenecks and generate graphical data, they are Gprof, and Gprof2dot, respectively.

Table 1 – Results from CLOC.

Language	Files	Blank	Comment	Code
Fortran 90	748	67905	126111	454676
C	14	2154	5102	6062
Bourne Shell	4	694	857	5081
C/C++ Header	39	390	1129	1496
make	18	186	173	891
Sum	823	71329	133372	468206

1.1 Objective

The main objective of this work is to show that is viable to migrate segments of code of BRAMS to a heterogeneous architecture, particularly hardware platforms that use Xeon Intel processor coupled to a programmable circuit (FPGA) via PCIe. Thus, we can provide a hardware/software solution for a snippet of BRAMS that will be functionally integrated to BRAMS.

According to the studies in the literature, the chemical equation from the mass continuity equation is the most computationally demanding part. This term calculates several linear systems $Ax = b$. Thus, we implemented such equations in hardware and provided a portable and highly parallel design in OpenCL language. The OpenCL framework also allowed us to couple our circuit to BRAMS legacy code in Fortran90. Although the development tools present several problems, the designed solution has shown to be viable with the exploration of parallel techniques. However, the performance was below of what we expected.

1.2 Motivation

The weather prediction is considered a super application or a supercomputing application. They demand high computing power with increasing inclination for growth (KIRK; WEN-MEI, 2012).

In the last 20 years, processors have increased in 1000 times their performance. Increasing the performance of current processors require changes in the architecture. Due to energy limits, it is not viable to boost the frequency on current state-of-art processors. This limitation has forced

designers to use large-scale parallelism, heterogeneous cores, and accelerators for demanding applications (BORKAR; CHIEN, 2011).

The applications will depend on customized accelerators, especially in hardware, to gain performance. In many cases, the acceleration also improves computational efficiency regarding the energy consumption compared to a solution based on software (CONG; ZOU, 2009).

Accelerators are special-purpose processors used to accelerate CPU-bound applications. The development of them is usually in Graphics Processing Unit (GPU) or Field-Programmable Gate Array (FPGA); both can achieve substantial performance for certain workloads when compared to Central Processing Unit (CPU). FPGAs are highly customizable and, in general, offer the best performance (CHE *et al.*, 2008).

1.3 Document organization

In Chapter 2, we present the fundamental concepts for this project and the literature review. In Chapter 3, we describe our three architectures for Jacobi method. We also describe how we managed the interoperability with CPU for each architecture. In Chapter 4, we present the profiling that supports that the chem term is the most expensive in the mass continuity equation. Then we show the results for each architecture and its problems. Lastly, we compare our hardware solution with the current software solution in BRAMS. In Chapter 5, we conclude our project and present the main limitations and future work. In Appendix A, we depict the installation of BRAMS. In Appendix B, we show how to visualize the results from BRAMS. In Annex A, there are useful links to libraries, additional software, and BRAMS source code.

FUNDAMENTAL CONCEPTS

2.1 Brazilian developments on the Regional Atmospheric Modelling System – BRAMS

BRAMS is a project originally developed by [ATmospheric, Meteorological, and Environmental Technologies \(ATMET\)](#), [IME/USP \(Institute of Mathematics and Statistics/University of São Paulo\)](#), [IAG/USP \(Institute of Astronomy, Geophysics and Atmospheric Sciences\)](#) and [CPTEC/INPE \(Center for Weather Forecasts and Climate Studies/National Institute for Space Research\)](#), and funded by [Financier of Studies and Projects \(FINEP\)](#) (Brazilian Funding Agency) ([INPE/CPTEC, 2015](#)).

They aimed at producing an adapted version of [Regional Atmospheric Modeling System \(RAMS\)](#) for the tropics ([FREITAS *et al.*, 2009](#)), which provided a single model to Brazilian Regional Weather Centers. One of the purposes of BRAMS/RAMs is to simulate atmospheric circulations through a numerical prediction model. The simulation can range from hemispheric scales down to large eddy simulations (LES) of the planetary boundary layer ([LONGO *et al.*, 2013](#)).

Since version 4.2, the CPTEC/INPE team is responsible for the entire software development. BRAMS uses the cathedral model. Software built in a cathedral model must provide the source-code every release, and only the software developers can access the source-code between releases ([RAYMOND, 2001](#)). The software license is under CC-GNU-GPL, and some parts may receive other restricted licenses.

Three main models represent BRAMS. The tracer transport model, chemical model ([Coupled Chemistry Aerosol–Tracer Transport \(CCATT\)](#)) and a surface model. BRAMS incorporate the tracer transport model and chemical model, and [Joint UK Land Environment Simulator \(JULES\)](#) is the name of the surface model. In this dissertation, we focus on CCATT, more specifically the numerical solution of the chemical reactivity.

2.1.1 CCATT-BRAMS

CATT-BRAMS is a Eulerian atmospheric chemistry transport model fully coupled to BRAMS. Its design allows us to study transport processes associated with the emission of tracers and aerosols (FREITAS *et al.*, 2010). CATT-BRAMS solves the mass continuity equation for tracers; we present it in Equation (2.1).

$$\begin{aligned} \frac{\partial s}{\partial t} = & \left(\frac{\partial s}{\partial t} \right)_{adv} + \left(\frac{\partial s}{\partial t} \right)_{PBL\ diff} + \left(\frac{\partial s}{\partial t} \right)_{deep\ conv} + \\ & \left(\frac{\partial s}{\partial t} \right)_{shallow\ conv} + \left(\frac{\partial s}{\partial t} \right)_{chem} + W + R + Q \end{aligned} \quad (2.1)$$

“Where s is the grid box mean tracer mixing ratio” (LONGO *et al.*, 2013); a prognostic variable, this variable is governed by the prognostic equation, which means that involve derivatives (RANDALL, 2013). “The term *adv* represents the 3-D resolved transport (advection by the mean wind); and the terms *PBL diff*, *deep conv*, and *shallow conv* stand for the sub-grid scale turbulence in the Planetary Boundary Layer (PBL), and deep and shallow convection, respectively”.

Advection and convection stand for the transfer of energy generated by the movement of particles of liquid like water in the atmosphere. Advection transfer horizontally, and convection transfer energy vertically (ACKERMAN; ACKERMAN; KNOX, 2013). Deep convection is the thermally driven turbulent mixing that lifts the air from the lower to the upper atmosphere. “Shallow convection: thermally driven turbulent mixing, where vertical lifting is capped below 500hPa” (DAVISON, 1999; VAUGHAN, 2009).

“The *chem* term refers simply to the passive tracers’ lifetime, the *W* is the term for wet removal applied only to aerosols, and *R* is the term for the dry deposition applied to both gasses and aerosol particles” (LONGO *et al.*, 2013).

CATT-BRAMS evolved to CCATT-BRAMS (Chemistry CATT-BRAMS). This new model includes a gas phase chemical module, which solves *chem* term in Equation (2.1). We show this module in Equation (2.2).

$$\left(\frac{\partial \rho k}{\partial t} \right)_{chem} = \left(\frac{d\rho k}{dt} \right) = P_k(\rho) - L_k(\rho), \quad (2.2)$$

Where ρ stands for the number density for each of the N species, and P_k and L_k are the net production and loss of species k , respectively. P and L terms include photochemistry, gas phase, and aqueous chemistry. The solution of this equation is the most expensive term of Equation (2.1).

The development of CCATT required advanced numerical tools to provide a flexible multi-purpose model, i.e. the model can run for both operational forecasts and research simulations. Figure 1 illustrates the simulation of CCATT-BRAMS system. The illustration represents the primary sub-grid scale processes involved in the trace gas and aerosol distributions.

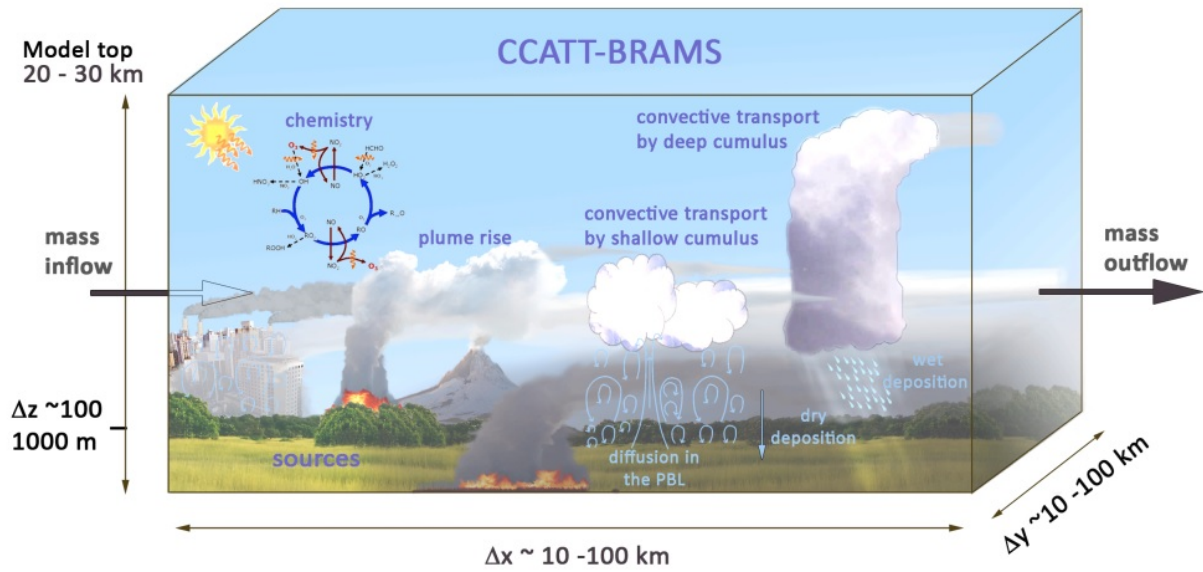


Figure 1 – Simulation of CCATT-BRAMS system, figure from (LONGO *et al.*, 2013).

Moreover, the model system allows the user to provide any chemical mechanism. Currently, there are three widely used chemistry mechanisms; they are as follows: **Regional Atmospheric Chemistry Mechanism (RACM)** with 77 species (STOCKWELL *et al.*, 1997), **Carbon Bond (CB)** with 36 species (YARWOOD *et al.*, 2005), and the **Regional Lumped Atmospheric Chemical Scheme (RELACS)** with 37 species (CRASSIER *et al.*, 2000).

Scientific projects frequently use RACM mechanism due to its number of species it covers; RELACS is a reduced version of RACM. CPTEC uses RELACS for operational air quality prediction. According to Gácita (2011), RELACS can replicate RACM results reasonably well.

To solve the Equation (2.2) with k species Longo *et al.* (2013) uses Rosenbrock method (WANNER; HAIRER, 1991; VERWER *et al.*, 1999) to change from nonlinear differential equation system to a linear algebraic increment in terms of K_i . In this method, the integration step is adjusted as a function of the calculated error (FERNANDES, 2014).

The solution of this linear algebraic increment, which corresponds to P and L, is in Equation (2.3).

$$\rho(t_0 + \tau) = \rho(t_0) + \sum_{i=1}^s b_i K_i, \quad (2.3)$$

Where t_0 stands for initial concentration, τ is the timestep. The product sum approximates the integral, where i is the Rosenbrock stage. Each timestep and stage require the update of K_i

increment according to the linear system in Equation (2.4a).

$$\left\{ \begin{array}{l} K_i = \tau F(\rho_i) + \tau J(\rho(t_0)) \cdot \sum_{j=1}^i \gamma_{ij} K_j \\ \rho_i = \rho(t_0) + \sum_{j=1}^{i-1} a_{ij} K_j \\ F(\rho_i) = P(\rho_i) - L(\rho_i), \end{array} \right. \quad \begin{array}{l} (2.4a) \\ (2.4b) \\ (2.4c) \end{array}$$

Where a_{ij} and γ_{ij} are constants that depend on s , ρ_i stands for the intermediate solution used for recalculating the net production on stage i given by the term $F(\rho_i)$, and J is the Jacobian matrix of the net production at time t_0 . Solving the Equation (2.5) is the most computing intensive.

$$Ax = b \quad (2.5)$$

Where A is a $N \times N$ matrix, N is the number of species. The vector x is the solution, and b is the right-hand side or vector of the independent terms. BRAMS solves the Equation (2.4b) by using Sparse1.3a (KUNDERT; SANGIOVANNI-VINCENTELLI, 1988). In Figure 2 we show the pseudo-algorithm for the Rosenbrock method with Sparse1.3a to solve each stage.

CCATT-BRAMS runs operationally at CPTEC/INPE since 2003; it covers the entire South America with a spatial resolution of 25 km. It is possible to predict the emission of Gases and Aerosols in real time¹, as well as meteorological variables² (MOREIRA *et al.*, 2013).

2.1.2 Libraries

In this subsection, we will briefly explain each library necessary for BRAMS.

2.1.3 NetCDF

According to Rew (2015) Network Common Data Form (NetCDF) “is a set of interfaces for array-oriented data access and a freely-distributed collection of data access libraries for C, Fortran, C++, Java, and other languages. The netCDF libraries support a machine-independent format for representing scientific data. Together, the interfaces, libraries, and format support the creation, access, and sharing of scientific data”.

This library creates an abstraction level for machine-dependent data representation. Such abstraction allows the application to share those files across the networks on different workstations. Programs with NetCDF interface can read and write data without the restriction of machine-dependent binary data files (REW; DAVIS, 1990).

¹ <http://meioambiente.cptec.inpe.br/>

² <http://previsaonumerica.cptec.inpe.br/golMapWeb/DadosPages?id=CCattBrams>

Algorithm: Rosenbrock Method

```

Input: Sparse1.3 data structure
1 begin
2   foreach block do
3     foreach grad_point do
4       Read variables from BRAMS;
5       Update photolysis rate;
6     Compute initial kinetic reactions;
7     while Timestep < threshold do
8       Compute Jacobian of the matrix of concentrations;
9       Compute Equation (2.2);
10      foreach chemical_specie do
11        foreach grad_point do
12          Update F( $\rho$ ) on the data structure;
13      while error > tolerance do
14        foreach chemical_specie do
15          foreach grad_point do
16            Update matrix A;
17        Update  $b_i$ ;
18        foreach grad_point do
19          Compute 1st Rosenbrock method;
20        Update  $b_i$ ;
21        foreach grad_point do
22          Compute 2nd Rosenbrock method;
23        Update matrix of concentrations  $\rho$ ;
24        Update production term F( $\rho$ );
25        Update  $b_i$ ;
26        foreach grad_point do
27          Compute 3rd Rosenbrock method;
28        Update matrix of concentrations  $\rho$ ;
29        Update production term F( $\rho$ );
30        Update  $b_i$ ;
31        foreach grad_point do
32          Compute 4th Rosenbrock method;
33        Update matrix of concentrations  $\rho$ ;
34        Compute error and rounding;
35        if tolerance - rounding > 1.0 then
36          Accept solution;
37        else
38          Compute the new integration step;
39          Update the integration step;
40      Update variables from BRAMS;

```

Figure 2 – Rosenbrock Method.

The NetCDF software was developed by Glenn Davis, Russ Rew, Ed Hartnett, John Caron, Dennis Heimburger, Steve Emmerson, Harvey Davies, and Ward Fisher at the Unidata Program Center in Boulder, Colorado, and many users also contributed to software development.

2.1.4 HDF5

With the [Hierarchical Data Format \(HDF5\)](#) technology suite is possible to organize, store, discover, access, analyze, share, and preserve diverse, complex data in heterogeneous computing and storage environments ([GROUP, 2011](#)).

HDF5 supports all types of digital data from any origin or size. This suite is useful for data collected from satellites, nuclear testing models, high-resolution MRI brain scans. Besides the data, HDF5 files also contain the metadata necessary for efficient data sharing, processing, visualization, and archiving.

According to [Fazenda et al. \(2012\)](#), BRAMS uses HDF5 to overcome data writing phase that was preventing scalability of BRAMS to 9,600 cores; in this work, they used a technique called disk-direct. Such technique was essential to perform I/O collective operations; these operations interpolate the sub-domains in a single file in an external memory.

2.1.5 Zlib

Zlib is a library for lossless data-compression for use virtually on any computer hardware and operating system. The data format from zlib is portable across platforms. Jean-loup Gailly and Mark Adler are responsible for Zlib creation ([GAILLY; ADLER, 2015](#)).

2.1.6 Szip

Szip is a data compressor for data from the sphere. For energy compression, it uses a Haar wavelet transform on the sphere. This transformation reduces the entropy of the data. After this transformation, it encodes with Huffman and run-length. Both compression algorithms are lossless and lossy ([MCEWEN; EYERS, 2011](#)).

2.1.7 Mpich

[Message Passing Interface CHameleon \(MPICH\)](#) is a portable implementation of the [Message Passing Interface \(MPI\)](#). Of the goals of MPICH is to provide efficient MPI implementation for different computation and communication platforms ([MPICH, 2015](#)).

MPICH is open source. It works on several platforms, including Linux (on IA32 and x86-64), Mac OS/X (PowerPC and Intel), Solaris (32- and 64-bit), and Windows.

2.1.7.1 MPI

Programming and debugging for parallel algorithms are much more complicated than programming for sequential algorithms. There are several models of parallel programming; they are as follows (NIELSEN, 2016):

- Vector supercomputers, which relies on [Single Instruction, Multiple Data \(SIMD\)](#);
- Multi-core machines with shared memory, which uses multi-threading;
- Clusters of computer machines with distributed memory.

The latter can include the first and the second parallel programming; this is the parallel programming paradigm suitable for MPI. Each node can execute a program using its local memory, and cooperation among nodes depends on sending and receiving messages (GROPP *et al.*, 2014).

Message Passing Interface is an [Application Programming Interface \(API\)](#). This API hides the fine details of implementation from the programmer, and it also provides portability and efficiency with a wide acceptance from academy and industry. This API works with most common sequential languages, i.e. C, C++, Java, Fortran and so on (KARNIADAKIS; KIRBY, 2003).

2.2 Linear Equation

According to [Anton and Rorres \(2013\)](#), [Larson \(2016\)](#), a linear equation in n variables $x_1, x_2, x_3 \dots x_n = b$ can be represented in the form of the [Equation \(2.6\)](#).

$$a_1x_1 + a_2x_2 + \dots + a_nx_n = b \quad (2.6)$$

A system with m equations in n variables is called linear system. In [Equation \(2.7\)](#) we present a general linear system of this form.

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n &= b_2 \\ \vdots & \qquad \qquad \qquad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n &= b_m \end{aligned} \quad (2.7)$$

A solution whose $x_1, x_2, x_3 \dots x_n$ satisfies every equation is called consistent. Otherwise, it is inconsistent.

2.3 Linear Solver

According to [Golub and Loan \(2013\)](#) and [Peng \(2013\)](#), there are two fundamental categories to solve linear systems: direct and iterative methods.

2.3.1 Direct Method - LU

In theory, direct methods return the exact solution after a finite number of operations; in practice, this is not possible due to rounding errors. [Lower Upper \(LU\)](#) decomposition, Cholesky, Gaussian elimination are the main algorithms from this category.

Currently, BRAMS uses LU decomposition to solve the linear systems. Such method is computationally expensive, since LU decomposition requires $\mathcal{O}(n^3)$ and solving through backward and forward substitution requires $\mathcal{O}(n^2)$ ([BINDEL; GOODMAN, 2006](#)). The library responsible for decomposition and substitution is Sparse1.3a.

Sparse 1.3 is a package of subroutines in C for solving large sparse systems of linear equations. This library manages the necessary memory for the sparse matrix by using linked-list representation; it also offers an interface for Fortran, which turned the integration to BRAMS much simpler. Its original purpose was for use in circuit simulators; it is also able to handle node and modified-node admittance matrices ([KUNDERT; SANGIOVANNI-VINCENTELLI, 1988](#)).

2.3.2 Iterative Method - Jacobi

Regarding the iterative solvers, they offer an approximate solution after an infinite convergence process. Those algorithms convergences to $x = A^{-1}b$. Although simple, Jacobi has a highly parallel nature. [Equation \(2.8\)](#) shows an instance of Jacobi for a 3×3 matrix ([GOLUB; LOAN, 2013; MORRIS; PRASANNA, 2005](#)).

$$\begin{aligned}x_1 &= (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11}, \\x_2 &= (b_2 - a_{21}x_1 - a_{23}x_3)/a_{22}, \\x_3 &= (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33}.\end{aligned}\tag{2.8}$$

The current solution of this method requires the solution of the previous iteration. In the first iteration, it is common to suppose that all variables from the linear system are zero. In [Equation \(2.9\)](#) we show this computation, assume that $x^{(k-1)}$ is the previous solution and $x^{(k)}$ is the new approximation; from this equation, it is clear that the main diagonal is nonzero.

$$\begin{aligned}x_1^{(k)} &= (b_1 - a_{12}x_2^{(k-1)} - a_{13}x_3^{(k-1)})/a_{11}, \\x_2^{(k)} &= (b_2 - a_{21}x_1^{(k-1)} - a_{23}x_3^{(k-1)})/a_{22}, \\x_3^{(k)} &= (b_3 - a_{31}x_1^{(k-1)} - a_{32}x_2^{(k-1)})/a_{33}.\end{aligned}\tag{2.9}$$

The general algorithm for Jacobi is in [Figure 3](#).

Algorithm: Jacobi Method

```

Input: Matrix A, Vector x, Vector b
Output: Vector x
1 begin
2   for i < n do
3     for j < n do
4       if i ≠ j then
5         sum = sum + Aij × xjk-1;
6     xik = (bi - sum) / aii;
7   return x

```

Figure 3 – Jacobi Method algorithm.

From this algorithm, it is possible to infer a parallel computation of each row i since there is no dependence among rows. Iterative methods require stopping criteria that can identify when the error is small enough; this is essential for time execution as well. In our algorithm we used vector norm as the stopping criteria, we present vector norm in [Equation \(2.10\)](#).

$$\|\mathbf{x}\|_p = (|x_1^p| + \dots + |x_n^p|)^{1/p} \quad (2.10)$$

In this case, we consider $p = 2$, which is the Euclidian norm (standard vector length). As we want to measure the Euclidian distance between two vectors, the current solution and the previous solution, we use [Equation \(2.11\)](#). This distance must be close to zero, which means that the solution converged.

$$\xi_{abs} = \|x^{(k)} - x^{(k-1)}\| \quad (2.11)$$

We implemented [Figure 3](#) and the stopping criteria in [Equation \(2.11\)](#) in hardware using [Open Computing Language \(OpenCL\)](#). Intel FPGA SDK ([Software Development Kit](#)) allowed us to implement hardware in FPGA by using the OpenCL framework.

2.4 OpenCL

Until 2004, programmers could improve software time execution by just changing to a processor with a higher clock frequency. When Intel CPUs reached 3.6Ghz ([TSUCHIYAMA et al., 2012](#); [MUNSHI et al., 2011](#)), cooling commodity microprocessors became impractical; in [Figure 4](#), we show the increase of clock rate and power ([PATTERSON; HENNESSY, 2012](#)).

From this point on, it was evident to the vendor that increasing clock rate was not possible anymore. That forced the vendors to invest their money and efforts to change the

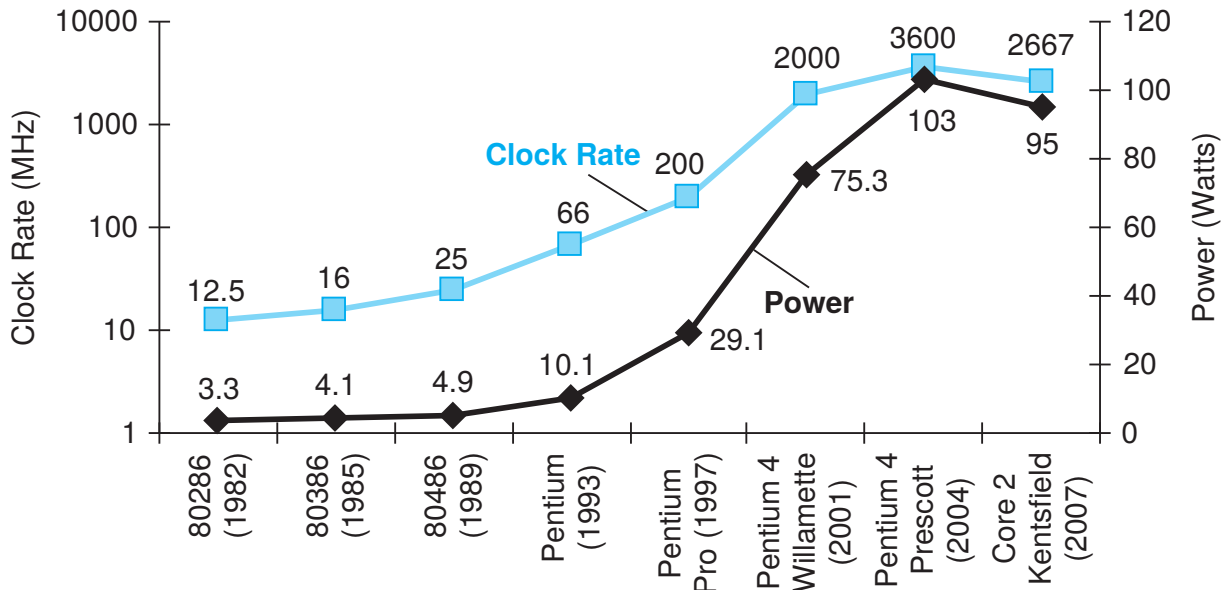


Figure 4 – Clock rate and power increase of eight generations of Intel microprocessors, figure from (PATTERSON; HENNESSY, 2012).

design of the processors; from 2006 until now, all desktop and server companies decided to ship multiprocessors per chip.

Current processors allow the programmer to improve throughput rather than response time. Most of the processors require parallel processing to take full advantage of them.

Although the most intuitive parallel programming is in CPU, it is possible to use parallel programming for accelerators; in this dissertation, we consider accelerator every non-CPU hardware. Shifting towards to multicore technologies imposes a severe change in software development, especially if there is heterogeneity of hardware (BUCHTY *et al.*, 2012).

Heterogeneous systems became critical for scientific and industrial applications, and OpenCL is the first industry standard for programming such systems. OpenCL supports a very wide range of systems, from smartphones to supercomputers; this framework delivers much more portability than any other parallel programming standard (MUNSHI *et al.*, 2011).

2.4.1 Data structures for OpenCL

Programming for heterogeneous platform demands the programmer to execute the following steps:

- Discovers the components in the heterogeneous system (CPU, FPGA, GPU);
- Retrieve the characteristics of these components; this allows the software to use specific features for each hardware component;
- Create the logic responsible for computing the problem on the platform;

- Establish the memory objects necessary for the computation;
- Define order execution of the kernels on the specific components of system;
- Gather the final results from the component.

We can accomplish such steps by using OpenCL API and its data structures. Every OpenCL application requires five data structures; they are as follows: device, kernel, program, command queue, memory object, and context.

The device, as the name says, is the set of accelerators available to perform some computation; the host is responsible for sending the data for computation. The kernel is the OpenCL function that performs the calculation on the device. The program is the source code or executable location responsible for implementing the kernels.

The API guarantee the order of memory transfers and kernel execution through the command queue. Memory objects maintain the necessary data (on the device) used by the kernels. Regarding the last data structure, we have the context; this structure conducts the interaction between the host and the kernels by managing all the previous data structures.

In [Figure 5](#), it is possible to see how data structures interact with each other. This picture represents OpenCL mapped to an FPGA device (green box), in this manner, program resides inside the FPGA.

These data structures are essential to guarantee OpenCL portability and programming model. OpenCL standard defines two different programming models: data-parallel and task-parallel programming model. Programmers must know both models when designing and application in OpenCL; defining which is better depends on the algorithm and the underlying hardware.

2.4.2 Data Parallelism

Data parallelism is suitable for SIMD, this kind of parallelism is the basis for GPU. Usually, this kind of model is perfect for matrix problems.

OpenCL API defines this programming model through [N-Dimensional Range \(NDRange\)](#). N ranges from one to three; each dimension must specify the index space extent. This index space range allows the programmer to divide the problem into work-groups and work-items.

In this author's opinion, programming using NDRange leads to a confusing index subdivision. Usually, the programmer learns that i stands for rows and j for columns. In this messy sea of indexes, we associate i to x and j to y ; which is the opposite of how OpenCL maps the index space. The first dimension, x , defines the width of the matrix, i.e. the dimension in columns. The second dimension, y , defines the size in rows.

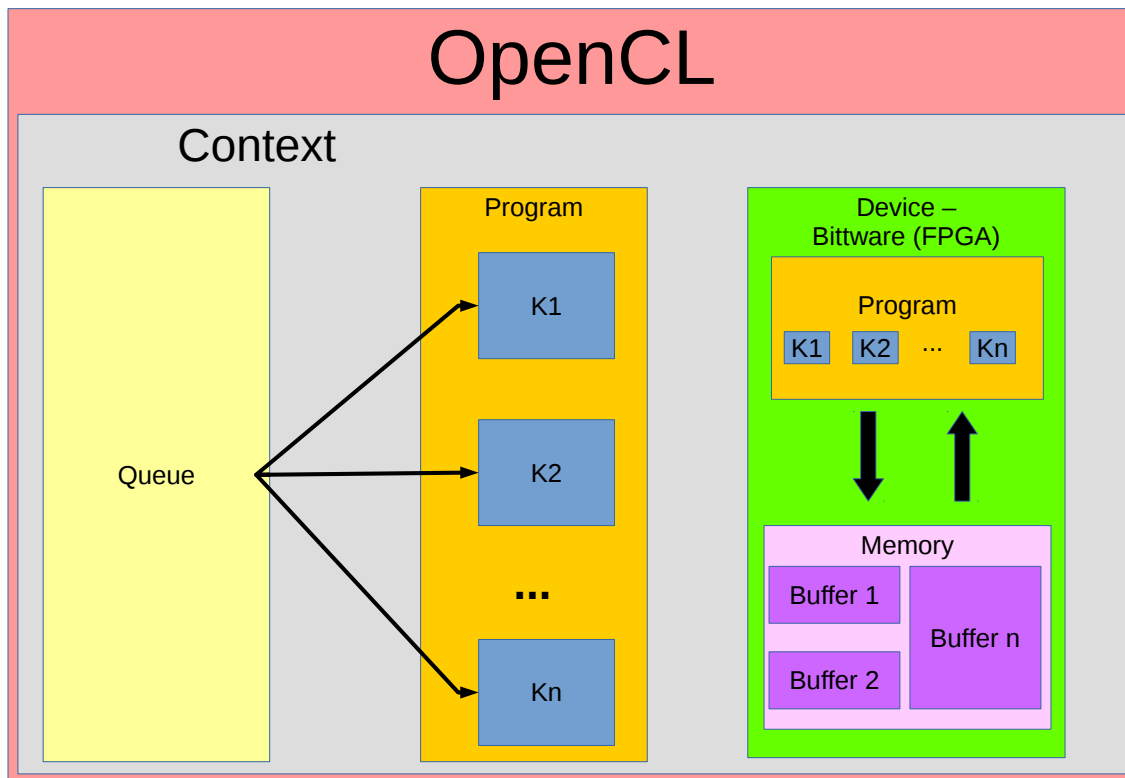


Figure 5 – OpenCL Data Structures – Consider Program as a single data structure; we replicated it to make the understanding easier.

This index space subdivision is the same for work-groups and work-items. A global problem can break into work-groups, and each work-group can have one or more work-items; we better explain this subdivision in [Figure 6](#).

By using work-groups, OpenCL API imposes some restrictions to the programmer. Only work-items that belong to the same work-groups can share data, which can impose dependencies on them. These dependencies require a work-group barrier synchronization. In OpenCL 1.0, synchronization is not possible between work-groups.

2.4.3 Task Parallelism

Although the OpenCL execution model aims at data parallelism as the primary target ([MUNSHI *et al.*, 2011](#)), the model also allows the programmer to use task parallelism. This parallelism uses a single work-item, this equivalent to NDRange defined as 1 for each dimension. According to [Tsuchiyama *et al.* \(2012\)](#), [Munshi \(2009\)](#) task parallelism is suitable when there are different commands; this application is common when using CPUs.

This kind of parallelism requires a method to balance the work between the processing units since a task can perform its work before the others. This parallelism is useful for pipelining, where multiple instructions execute at the same time in different stages of the pipeline; it is a

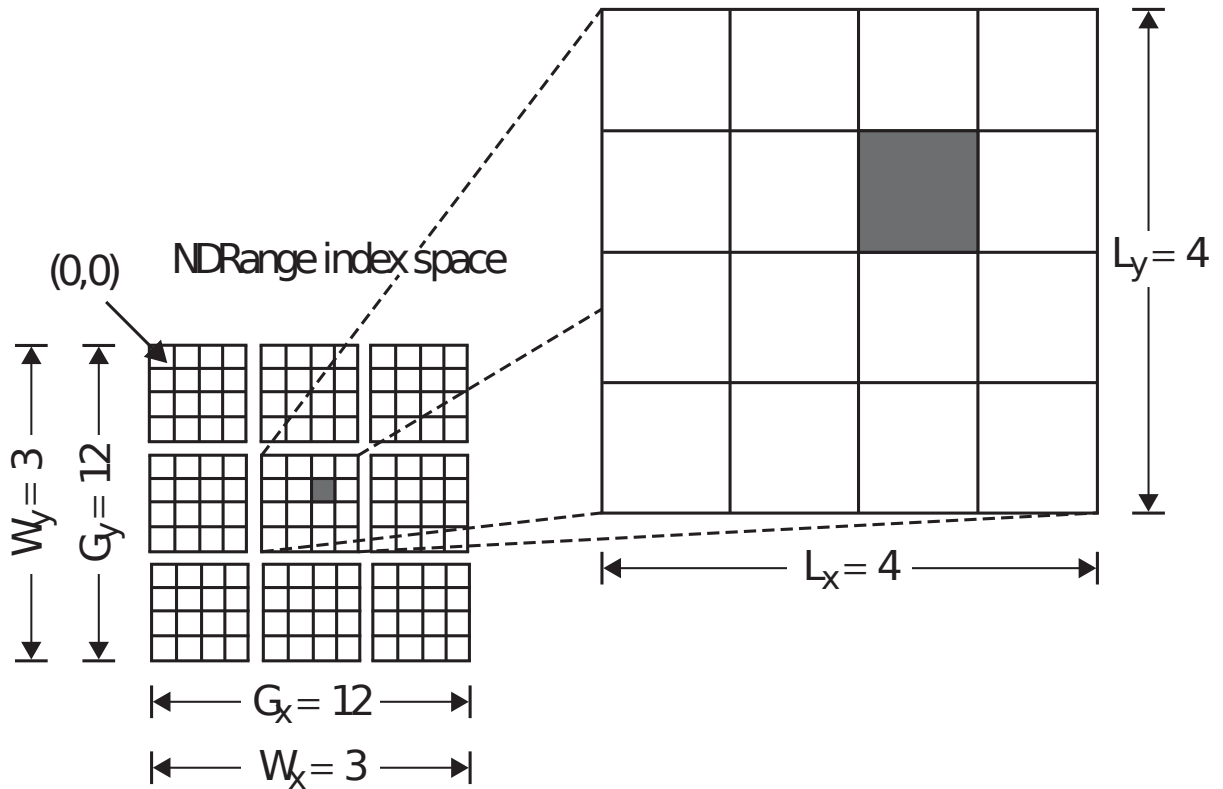


Figure 6 – An example of how the global IDs, local IDs, and work-group indices are related for a two-dimensional NDRange. For this figure, we have the following indices: the shaded block has a global ID of $(g_x, g_y) = (6, 5)$, a work-group ID of $(w_x, w_y) = (1, 1)$ plus a local ID of $(l_x, l_y) = (2, 1)$, figure from (MUNSHI, 2009).

crucial feature considering FPGA devices. Note that we did not mention GPUs; these devices, as we mentioned earlier, are suitable for data parallelism due to the number of cores available.

2.5 Intel FPGA SDK for OpenCL

Programming in OpenCL for CPU, GPU, ARM or FPGA requires the vendor to implement and provide for the programmer. In the scope of this dissertation we used Intel implementation for OpenCL in FPGAs.

By using OpenCL standard, we could abstract away the FPGA design. Debugging is also another important leading factor, it is possible to guarantee correct functioning of the kernel by emulating in the CPU. In this section, we present key features of this standard applied to FPGAs from Intel. In Figure 7 we show OpenCL system implementation on the FPGA.

In this figure, we present multiple kernel pipelines; a kernel represents a high-performance implementation of a hardware circuit (CZAJKOWSKI *et al.*, 2012a). Each of these pipelines connects to internal and external interfaces to memory (Figure 8 shows the partitioning of the FPGA). The external interface is necessary for accessing the Global Memory, which in turn requires a global interconnect to manage the request from different pipelines; this global interconnect is also needed for Peripheral Component Interconnect Express (PCIe) interface with the

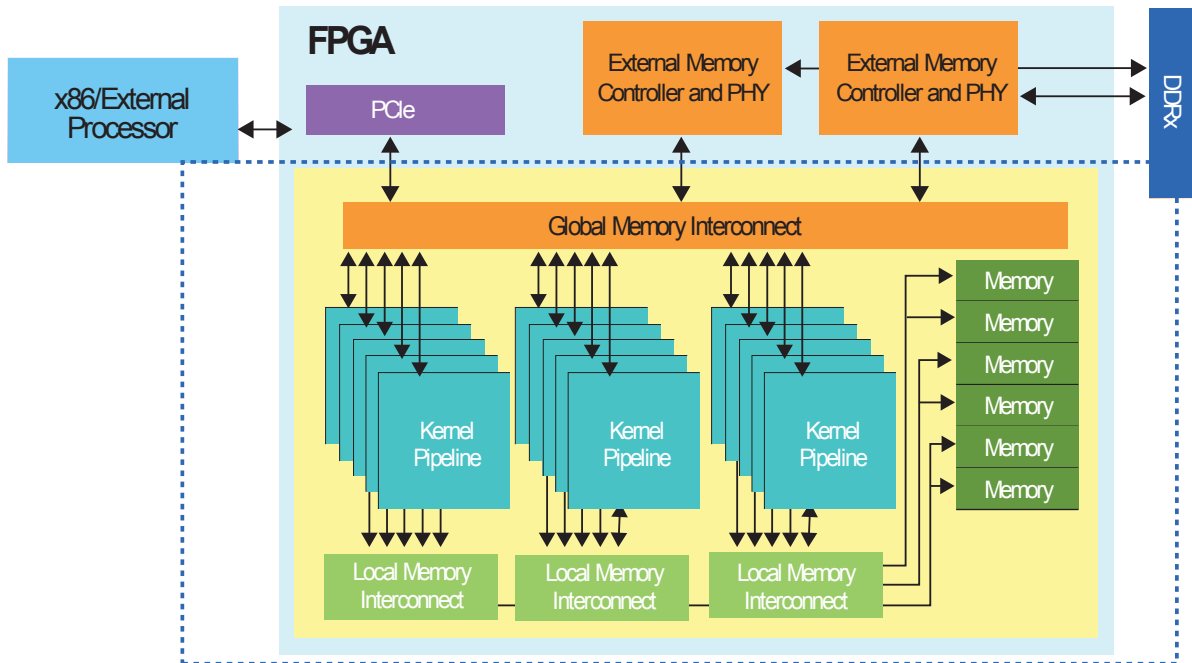


Figure 7 – Components from OpenCL system on Intel FPGAs, figure from (ALTERA, 2013).

host. the internal interface is critical to local memory (ALTERA, 2013; INTEL, 2016b).

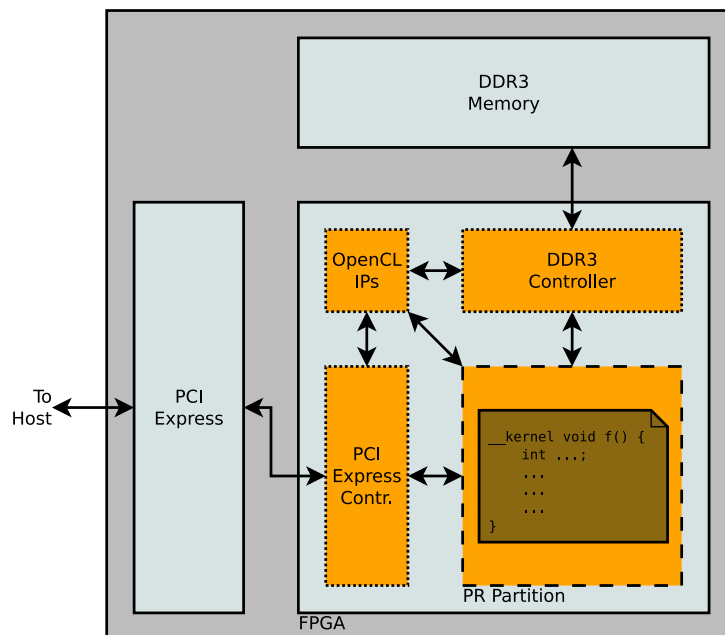


Figure 8 – Partitioning of the FPGA. PCIe, DDR3 controller and IPs are every project of OpenCL, so only the remaining is available for the kernels, figure granted by André Perina.

Unlike the GPU, where there are multiple cache levels, in FPGA local memory requires M20K blocks spread over the board (INTEL, 2016a). In Figure 9, we show a local memory with a single bank and three M20K blocks.

Regarding private memory, Intel uses FPGA register to implement them. That is the fastest memory in the hierarchy, and there is a generous number of them in the FPGA. The device can access these register in parallel, which allows a much higher bandwidth than any other

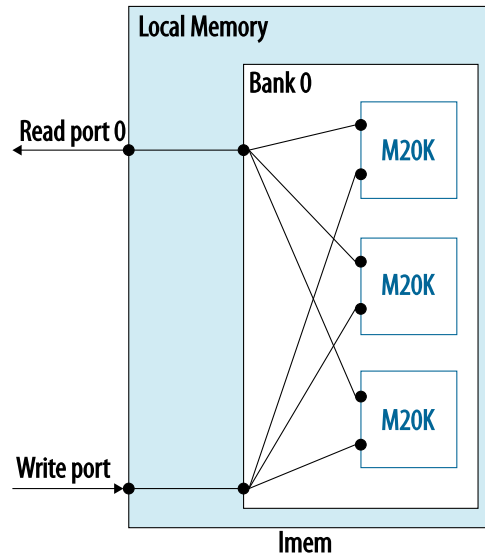


Figure 9 – Implementation of local memory with three M20K blocks, figure from (INTEL, 2016a).

memory in OpenCL. According to our experiments, Intel infers registers for single variables or small arrays; a relative big array requires a local memory.

Intel performs several optimizations before generating the hardware, and Figure 10 shows the flow of the compilation of OpenCL based on LLVM compiler infrastructure. The input is an OpenCL application (.cl) that contains a set of kernels and a host program (.c) (CZAJKOWSKI *et al.*, 2012b).

Compilation of the host source code uses a standard C compiler. The compiled file links with Altera OpenCL (AOCL) Host Library. Regarding the kernel source code, it uses an offline kernel compiler (JANIK; TANG; KHALID, 2015), i.e. the programmer must compile the kernel separate from the host; this process may take hours to compile.

Compilation of the hardware is not as simple as it seems. A C-language parser outputs an LLVM Intermediate Representation (IR) for each kernel (in essence, kernel is a C code); this intermediate representation is in the form of instructions and dependencies between them.

From this IR, the compiler optimizes it (live-value analysis) for an FPGA target. After optimizing, a Control-Data Flow Graph (CDFG) conversion takes place. The conversion is necessary to improve performance, reduce area and energy consumption before RTL generation (RTL generator) in Verilog for a kernel.

A system with interfaces to host and off-chip memory instantiates the compiled kernels. The host interface allows the host to access each kernel and specify workspace parameters and kernel arguments. Off-chip memory represents the global memory for a kernel in OpenCL, in our case, it is a DDR3 memory. Finally, we can synthesize the complete system in Figure 7 on an FPGA.

At last, the compiled host program has two elements. The first is the ACL Host Library;

it calls the functions that allow the host application to exchange information with the FPGA kernels. The second is the auto-discovery that allows the host program to detect the kernels types on an FPGA.

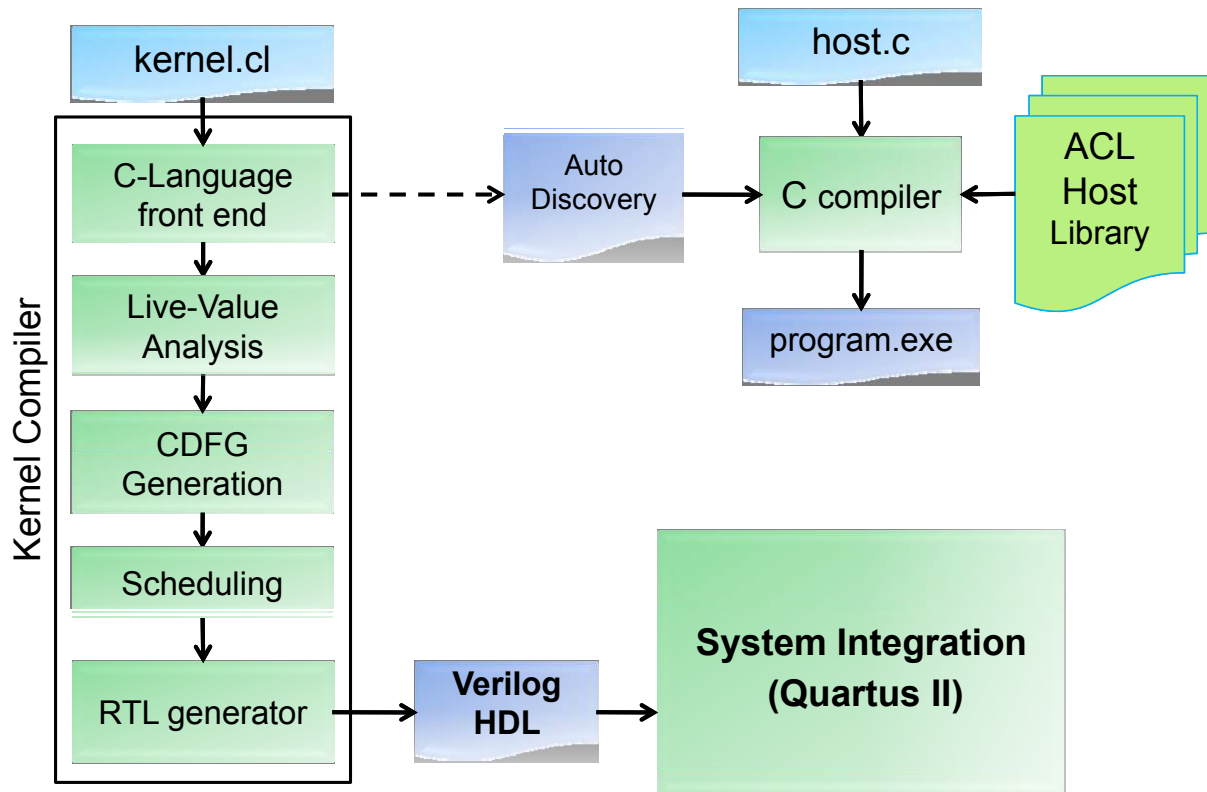


Figure 10 – Design flow with OpenCL, figure from (CZAJKOWSKI *et al.*, 2012b).

The main advantage of OpenCL over the traditional **Hardware Description Language (HDL)** is to produce designs with proper functionality without the FPGA design effort (considering the kernel is working correctly). Once the user has created a functional model, the focus is on the optimization. It is different from the HDL designs, where only in the design process we can assure the correct functionality (JANIK; TANG; KHALID, 2015).

We present further implementation details in [Chapter 3](#); we explain the pipelines of our architectures, memory hierarchy, interoperability between the host and the device.

2.6 FPGA

In 1985, Xilinx introduced the FPGA (BOBDA, 2007). An FPGA is a semiconductor device, which contains a two-dimension array of generic logic cells and programmable switches (CHU, 2011; MOORE ANDREW; WILSON, 2017). Over the years, the FPGAs has shown that capacity (number of gates) and speed are inversely proportional to price and power consumption, see [Figure 11](#).

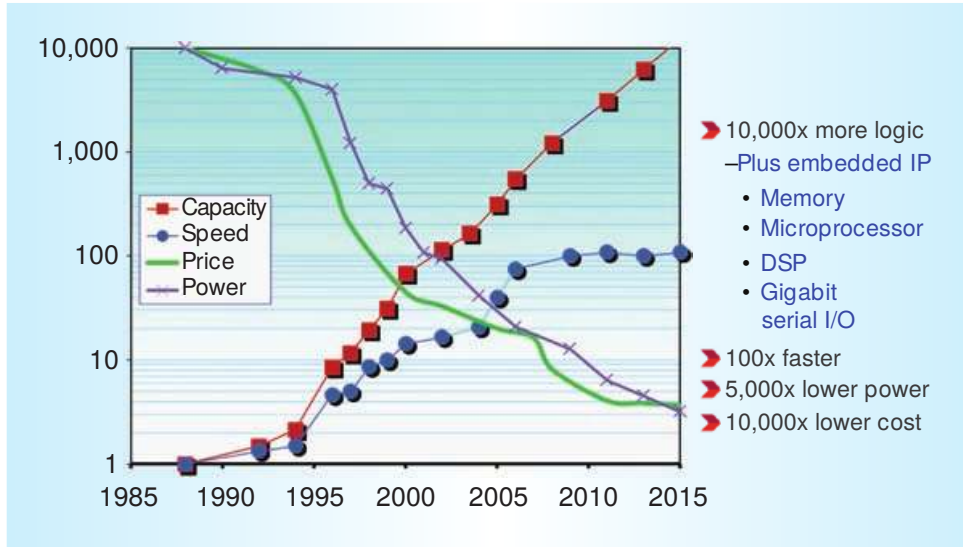


Figure 11 – Xilinx field-programmable gate array (FPGA) progression. (Price and power are per logic cell.), figure from (AHMAD *et al.*, 2016).

With an FPGA the programmer can define the behavior of the hardware after the manufacturing, that is why the name field programmable. That is possible due to the logic cells that can perform the behavior of different functions; once defined the logic and synthesized, the programmer can download the design through a bus to the FPGA; this bus can be a simple USB cable (BOUT, 2011).

Modern FPGAs contain a set of configurable [Static Random Access Memory \(SRAM\)](#), high-speed input/output pins (I/Os), logic blocks, and routing. They also have many [Logic Elements \(LE\)](#), which are the smallest unit of logic; usually, they are a [Look-Up Table \(LUT\)](#). Each LE can perform complex functions or simple basic logic as AND/OR. FPGAs also have configurable memory blocks; these memory blocks allows the programmer to provide a higher throughput since they are over the board.

Although FPGAs are reconfigurable, they also provide hard logic or hard [Intellectual Property \(IP\)](#), i.e. that does not change. Those circuits implement specific logic considered commodity, which allows the programmer to reduce cost and power of the design. These features allowed the programmers to build complex systems called [System-On-a-Programmable-Chip \(SOPC\)](#).

In general, SOPC or [System-On-Chip \(SOC\)](#) focus on lower-power electronics or high-performance applications. According to Silva (2014), SOPC is a suitable option for high-performance computing.

2.7 Development Environment for OpenCL

We used Bittware S5PH-Q-A7 board available at the [Reconfigurable Computing Laboratory \(LCR\)](#) at USP. In the next subsection, we further detail the main resources in this

device.

2.7.1 BittWare Board S5PH-Q

S5PH-Q is a half-length x8 card based on the high-bandwidth and the power-efficient Altera Stratix V GX A7. Stratix V FPGA is suitable for high-end applications; according to (DINIGROUP, 2017), this board guarantees 5,410 millions of gates for use.

The S5PH-Q is a versatile and efficient solution for high-performance network processing, signal processing, and data acquisition (BITTWARE, 2015). LCR purchased this board for the projects involving BRAMS and heterogeneous computing. Figure 12 shows the block diagram for this device, in Table 2 we detail the features.

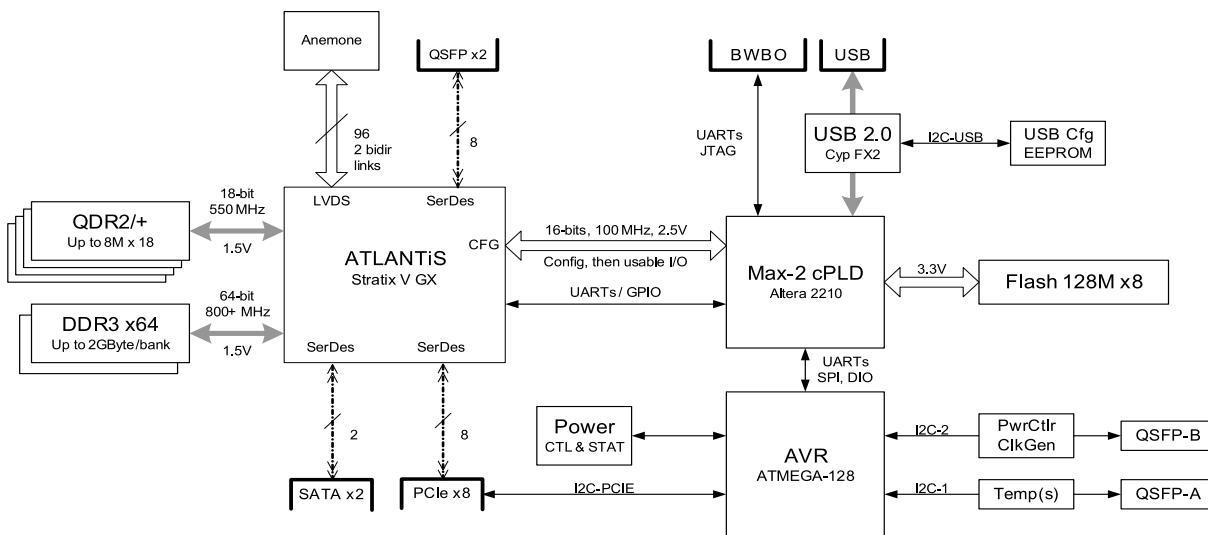


Figure 12 – Top-Level Component Implementation Block Diagram, figure from (BITTWARE, 2015).

2.8 Hardware/Software Codesign

Hardware/software codesign emerged in the 90's as a discipline, however this task was already common among the microprocessor companies. At the time, they were not conscious of the term codesign. Currently, a successful electronic system design requires the use of hardware/software codesign techniques (TEICH, 2012).

The current technology allows the programmers to deal with multiple processor cores, memory arrays, application specific hardware on a single chip (GALLERY, 2015). A more recent approach is from Intel on the HARP (Heterogeneous Architecture Research Platform) program, which included an Intel microprocessor and a Stratix V FPGA (GUPTA, 2015).

Such evolution in the technology requires the programmers to have knowledge in hardware and software, thus they can define the design trade-offs. In this manner, hardware/software codesign is becoming an ordinary task. In the literature we have some definition for hardware/software codesign.

Table 2 – S5PH-Q Features

Device	Features
Altera® Stratix® V GX FPGA	<ul style="list-style-type: none"> • 20 full-duplex, high-performance, multi-gigabit SerDes transceivers @ up to 14.1GHz • 952,000 logic elements (LEs) available • Up to 52 Mb of embedded memory • 1.4 Gbps LVDS performance • Up to 1,963 variable-precision DSP blocks • Embedded HardCopy Blocks
Memory	<ul style="list-style-type: none"> • Two banks of 4 GBytes DDR3 SDRAM (1Gx64) • Four banks of up to 18 MBytes QDRII/QDRII+(8M x 18) • 128 MBytes of Flash memory for booting FPGA
PCIe Interface	x8 Gen1, Gen2, Gen3 direct to FPGA
USB	USB 2.0 interface for debug and programming
Debug Utility Header	<ul style="list-style-type: none"> • RS-232 port to Stratix V • JTAG debug interface to Stratix V
QSFP+ Cages (optional)	2 QSFP+ cages on front panel connected to FPGA via 8 SerDes
Size	Half-length, standard-height PCIe slot card

According to [Schaumont \(2012\)](#), “Hardware/Software codesign is the design of cooperating hardware components and software components in a single design effort.”. Another definition in the book is: “the activity of partitioning, where one partition holds the flexible part (software), and the other the fixed part (hardware)”.

[Gallery \(2015\)](#) defines as a “concurrent design of both hardware and software of the system by taking into consideration the cost, energy, performance, speed and other parameters of the system”.

Figure 13 shows the pros and cons of Hardware and Software. In Hardware, it is possible to have a better performance, less energy consumption (more work done per unit of energy), power density (processors can no longer increase clock). In Software, design complexity is much harder in hardware, design cost, shrinking design schedules (time-to-market is reducing over the years, but software development can start even without a hardware platform).

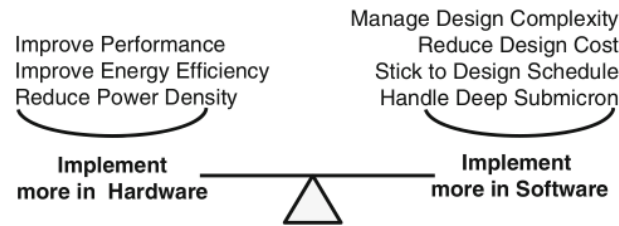


Figure 13 – Driving factors in hardware/software codesign, figure from (SCHAUMONT, 2012).

Partitioning or balancing is a hard task, and there is no magical solution. This work requires experience; another important factor is the cost, many times it is better a cheaper product than a fast product. [Blickle, Teich and Thiele \(1998\)](#) shows a theorem proving that determining a feasible allocation is a NP-Complete problem. In this work they consider the problem of mapping a set of tasks to resources.

There are some developed works on the literature that consider the hardware/software partitioning. [Gupta and Micheli \(1993\)](#) automates the design space exploration; initially, the algorithm of this work considers that all functionalities are in hardware and gradually moves to some of them to software based on the communication overhead. The problem is that much of the initial problem requires many resources from the hardware, because the initial guess starts from the problem entirely in hardware.

[Ernst, Henkel and Benner \(1993\)](#) follows an opposite approach, they start with an initial partition in software and gradually moves the software part into hardware. They used a partitioning heuristic, where the algorithm minimizes the amount of hardware resources. They show good results for the partitioning of the digital control of a turbocharged diesel engine and a filter algorithm for a digital image compared to software.

2.9 Related Work

Sparse matrices are necessary for several scientific and engineering applications. For example, least squares problems, eigenvalue problems, and image reconstruction. When compared to dense matrices, sparse matrices tend to be slower. The irregularity of memory access causes many cache misses, and there is the fact that memories are still much slower than processors. Another source for this slowness, is the high ratio of load and store operations, stressing the load/store units ([ZHUO; PRASANNA, 2005](#)).

FPGAs can effectively compute sparse matrix. The modern FPGAs provide multiple spatial floating-point operators, a significant of high-bandwidth on-chip memory, and abundant I/O pins ([KAPRE; DEHON, 2009](#)).

[Kapre and DeHon \(2009\)](#) parallelizes a Sparse Matrix Solver for SPICE (Simulation Program with Integrated Circuit Emphasis); the results range from 300 to 1300 MFlop/s on a Xilinx Virtex-5, while the processor (Intel Core i7 965) achieved 6 to 500 MFlops/s. According

to the authors, the former library (Sparse 1.3a) was not suitable for parallelization on FPGAs due to the frequent change of the non-zero pattern of the matrix.

They used the KLU solver. Circuit simulations are suitable problems for this solver. According to [Eller, Singh and Sandu \(2010\)](#), LU decomposition is not easily parallelizable. Later, they integrated the solver to SPICE ([KAPRE; DEHON, 2012](#)). In this new version, they also studied the energy savings of their work, which ranges from $8.9\times$ up to $40.9\times$ compared to CPU. They provide a codesign between the MicroBlaze and their hardware; MicroBlaze has poor support for double precision.

LU direct method emerged at the beginning of 2000 with [Daga et al. \(2004\)](#), [Zhuo and Prasanna \(2006\)](#). However, none of the works consider sparse matrices; which is the problem of BRAMS. In [Wu et al. \(2011\)](#), they compute the preprocessing in CPU and the numeric factorization in FPGA of the LU algorithm; they use sparse representation.

In [Foertsch, Johnson and Nagvajara \(2005\)](#), they consider the problem of Full-AC load flow, an important task in power system analysis. In their work, they compare Jacobi method to Newton-Raphson methods and conclude that Jacobi could outperform Newton method by exploring pipeline parallelism in FPGA. They do not consider any coupling to the load flow problem, i.e. there is not codesign.

[Morris and Prasanna \(2005\)](#), [Prasanna and Morris \(2007\)](#) study a related problem to BRAMS, they solve [Partial Differential Equations \(PDE\)](#) discretized in a linear system (sparse and dense) in FPGA. They also consider a highly pipelined Jacobi; the algorithm represented in floating point with 64 bits. However, they fail to consider a better approach to fit bigger matrices, each column of the row requires a multiplier. In this manner, hardware resources are proportional to matrix size.

When they need bigger matrices, they multiplex the entries to the underlying hardware. They also have problems with reduction, which required the implementation of an efficient one. The authors do not provide any circuit to test the convergence of the algorithm; they test their hardware with a predefined number of iterations.

[Kasbah and Damaj \(2007\)](#) implements Jacobi method in Handel-C. According to them, the hardware of the method can outperform the same algorithm in Software. They validate their tests by using the Handel-C simulator, i.e. they do not perform any test on the hardware. The authors also had to consider the algorithm in integer representation since Handel-C does not support floating point, and the library from Celoxica had some bugs to handle more than four floating point operations. In this dissertation, we do not simulate the results; all the computations are in FPGA, which provides much more accurate results.

[Bravo et al. \(2006\)](#) proposes the use of Jacobi method to solve the Eigenvalue and Eigenvector problems; a similar work is in [Wang and Wei \(2010\)](#). In the proposed architecture, they improve FPGA area by implementing the whole system in VHDL. They compare their

results, represented in floating point with 18 bits, with the results of the CPU, represented in floating point with 64 bits, and conclude that their design is faster and provides accurate results. In this work there is no codesign, the entire execution is in hardware with pre-fixed values stored in ROM memory.

Ruan *et al.* (2013) presents a similar approach that we used in BRAMS; they use a high-level synthesis from Maxeller, where the programmer defines the kernel in Java and MaxCompiler is responsible for creating a bitstream file. They also provide a codesign between CPU and FPGA; in their project, they provide a modification for Jacobi method called pipeline-friendly Jacobi.

According to their results, MaxCompiler generated a hardware that is capable of running at 175Mhz on Virtex-6. They compared their results with three different configurations: FPGA v.s single-thread CPU, FPGA v.s multi-thread CPU, and FPGA v.s MPI CPU. We show their speedup results in Figure 14, as we can see FPGA is superior even with MPI parallelism. Matrix size is a problem in their design, they cannot fit matrices bigger than 200×200 in the memory.

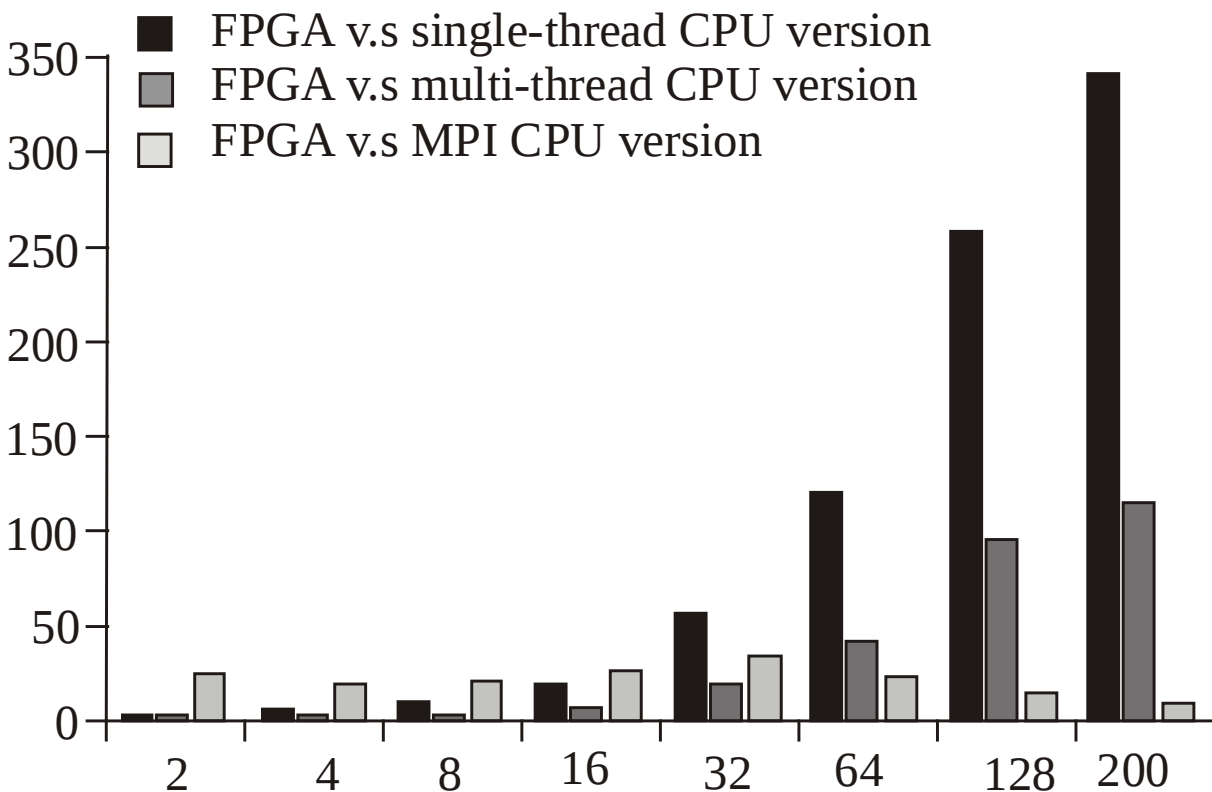


Figure 14 – Speedup compared to CPU versions. The x dimension stands for matrix size, and y dimension speedup in FPGA.

Schmid *et al.* (2014) proposes the use of the [Heterogeneous Image Processing Acceleration Framework \(HIPA^{cc}\)](#) for [Multiresolution Analysis \(MRA\)](#). They use this framework to generate code for the FPGA target; one of the case studies proposes Jacobi method as a smoother for PDE. They achieved 154Mhz on a mid-range FPGA (Xilinx Zynq 7045).

Although the success regarding the results, their design suffers from the number of block

RAMS. They cannot control the underlying hardware to reuse block RAM more efficiently. The pragmas in the framework have a severe impact over the [Initiation Interval \(II\)](#) ([SCHMID *et al.*, 2014](#)); however the framework does not offer any tool to help the programmer improve II.

[An and Wang \(2016\)](#) uses OpenCL for programming [Singular Value Decomposition \(SVD\)](#) in the AMD GPU. For computing SVD they used an adapted version of Jacobi method called one-sided Jacobi method. According to [Lambers \(2010\)](#), one-sided Jacobi implicitly applies the Jacobi method for the symmetric eigenvalue problem.

In their work, they used a W9100 graphics card from AMD, and according to the authors, it was the best and fastest graphics card. They performed the same tests we did, i.e. they use single and multi-thread to program the kernel. For small matrices, they had a better execution time single thread, for matrices with the order of 16 they had an improvement by using multiple threads. In this work, they do not perform any study related to power consumption, according to AMD vendor, this board can consume up to 275W.

In [Gomes \(2009\)](#), they use Jacobi for fluid simulation by implementing the algorithm in CUDA. According to them, the algorithm suffers from global memory latency. In FPGA we could improve this problem by using more local memory during computation since we can define our local memory size (as long as it does not exceed the hardware capacity). We avoided such problem due to the small memory footprint of our problem.

[Fernandes \(2014\)](#) decided to solve the linear systems from chemical reactivity of BRAMS by using optimum linear estimation. The author implemented the algorithm in OpenMP and OpenACC, although they achieve good results in time execution and precision, they do not couple their algorithm to BRAMS. According to [Fazenda *et al.* \(2006\)](#), coupling the chemical reactivity to CCATT-BRAMS is a complex task.

[Michalakes and Vachhrajani \(2008\)](#) developed [Kinetic PreProcessor \(KPP\)](#) Rosenbrock chemical integrator for GPUs in CUDA. They converted WRF Single Moment 5-tracer (WSM5) to CUDA, a model that represent the microphysics of clouds and precipitation. Their GPU did not provide double precision, so they needed to use single precision. Their implementation also requires a CPU per GPU. Their results show a speed-up of $1.23\times$, but they do not mention if this is due to the lower precision of the results.

[Linford and Sandu \(2009\)](#) used a [Cell Broadband Engine Architecture \(CBEA\)](#) to solve mass balance equations of Chemical transport model. They implemented a 3D chemical transport module for FIXED-GRID, this model also uses Rosenbrock method. Their work presents a superlinear speed-up. However, their *vector stream processing* requires the kernel to process the double of computing that it needs.

According to the authors these useless arithmetic operations is crucial to sustaining a higher throughput; otherwise branching conditions would be prohibitively expensive. Branching is not a problem on FPGA since all code-paths are established in hardware ([ACCELEWARE](#),

2014).

Eller, Singh and Sandu (2010) also use Rosenbrock method to solve the chemical system. In their experiments, they use Rodas-3 and Rodas-4 methods. Although the names suggest the number of stages, Rodas-3 uses four stages and three function calls. Rodas-4 has six stages and five function calls. They had to make significant changes in Rosenbrock data structure to fit the data in a 16KB of local memory. Memory transfer is responsible for one-third of the overall running time.

According to the authors, Rosenbrock does not provide any speed-up for each iteration; in some cases, the results in GPU was slower. Their results point to the fact that local memory is the main problem.

In Fu *et al.* (2017), they implement Shallow Water Equations solver in three different architectures. One of them is CPU-FPGA, they had to decompose the solver in three FPGAs to fit the double precision version; this solution resulted in a low bandwidth among the FPGAs. A second approach involved the use of mixed precision, which resulted in a relative error of less than 2% and 80% of the resources of a single FPGA; they implemented their design by using Java from MaxCompiler. This version had a speedup of $75\times$ compared to CPU.

In Yang *et al.* (2016), they present an ultra scalable solution — in the order of 10M-core — for Fully-Implicit Solver for Nonhydrostatic Atmospheric Dynamics. They implement a highly parallel incomplete LU method that does not sacrifice convergence rate in their SW26010 processor.

In Table 4, we provide the main features of each work compared to our third architecture Arch 3.

Table 4 – Table of comparison among related works

Features	A	B	C	D	E	Advantages	Disadvantage
Arch 3	X	X	X	X	X	100% in hardware.	Performance loss due to data representation.
Kapre	X	X		X	X	Direct Method.	Floating point division impacts on the performance.
Nagvajara	X	O	X			Compare Jacobi to other iterative methods.	It does not test in hardware.
Prasana	X	X	X		X	Division latency hidden.	Hardware scale according to matrix size.
Kasbah	X		X			Fast prototype with Handel-C.	Integer representation.
Bravo	X		X		X	It requires less FPGA area.	VHDL.
Ruan	X	O	X	X	X	High-level synthesis with Java.	Java is not a parallel programming language.
Schmid	X		X		X	Optimized code for different architectures.	Block are not reused.
An	X		X	X		Portable code.	Intense communication CPU-GPU.
Gomes	X	O	X	X		First CUDA implementation. Comparison between different iterative solvers.	Nvidia GPU-only.
Fernandes	X	X				Optimum linear estimation has faster conversion.	Not coupled to BRAMS.
Michalakes	X			X		WSM5 is 100% in GPU.	GPU board without double precision support.
Linford	X	X		X		Superlinear speed-up	Half of the computation is not necessary.
Eller	X	X		X		Comparison with CPU and GPU with parallel paradigm.	Memory transfer is responsible for one-third of the overall running time.

Features:

- A: Parallel;
- B: Double Precision;
- C: Iterative Method;
- D: Codesign;
- E: Hardware;
- O: No information about it.

DEVELOPMENT OF THE CODESIGN FOR THE CHEMICAL REACTIVITY OF BRAMS

Currently, BRAMS bases on software and MPI parallelism. This project focus on the chemical reactivity term included in BRAMS (CCATT). According to [Zhang *et al.* \(2011\)](#) and [Linford *et al.* \(2009\)](#), chemical reactivity can represent 90% of the computational time. On Tupã (Cray XE6) CPTEC/INPE this term is responsible for 80% of computational time ([FERNANDES, 2014](#)). In this chapter, we compare our approach with the implementation of BRAMS for solving the linear systems from the chemical reactivity.

3.1 CCATT–BRAMS software

BRAMS solves the mass equation of the chemical reactivity term using Sparse1.3a. This library in C decomposes [Equation \(2.5\)](#) into LU form. [Figure 15](#) represents BRAMS (gray square) and Sparse1.3a (yellow square); it is important to point out that we used BRAMS with a single grid.

We divided [Figure 15](#) into three regions for a better explanation of how chemical BRAMS couples with the reactivity. The blue outermost region represents the South America, according to [Alisson \(2016\)](#) the Brazilian model can apply the chemical reactivity to the South America with a 20km resolution and 3.5 days of anticipation. The grayish middlemost region exemplifies BRAMS over the South America with only a single process, i.e. one process is responsible for computing the entire grid. The following yellowish inner region represents Sparse1.3a library inside the process. In [Figure 16](#), we represent BRAMS with multiple processes, note that we still have a single grid.

Each MPI process computes its slice of the domain decomposition. The shaded areas are the ghost zones; these regions are necessary for solving the fluid dynamical equations, as a consequence, the nearest neighbors generate communications among them. The strongest

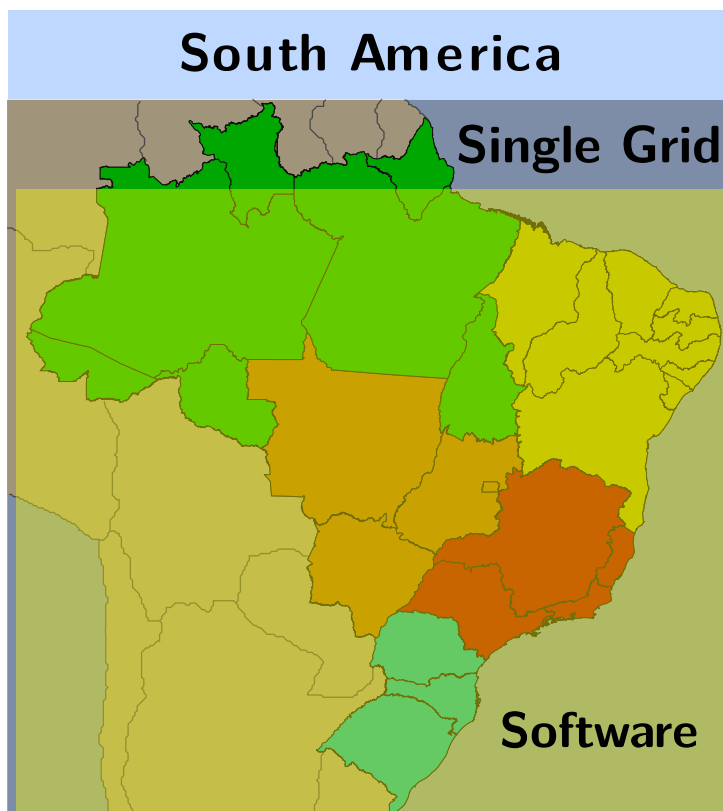


Figure 15 – Generic representation of BRAMS system with a single process. In this figure, we present BRAMS over the South America with a single grid, yellow square represents Sparse1.3 running for all the points over the grid.

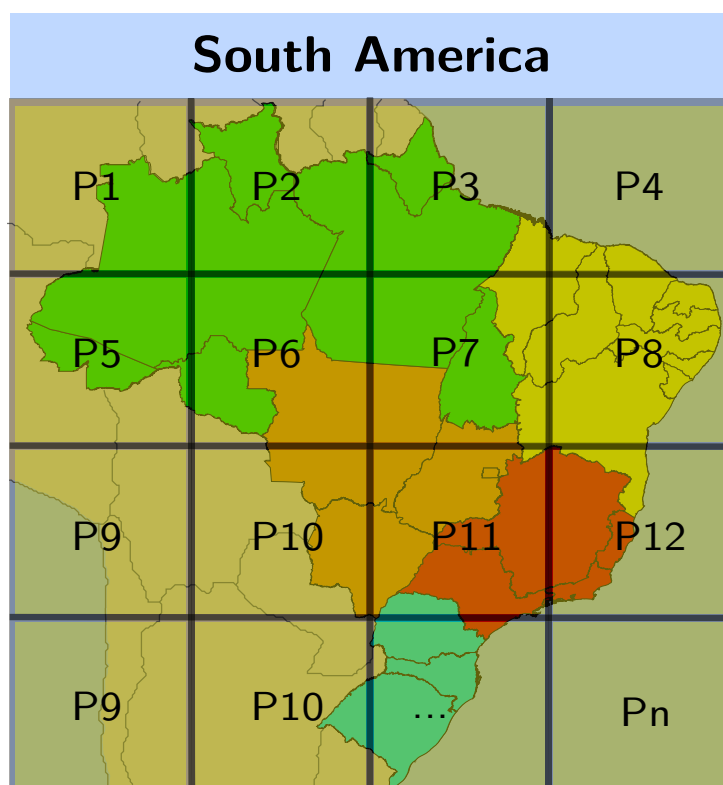


Figure 16 – Generic representation of BRAMS system with MPI processes. In this figure, we present BRAMS over the South America with a single grid distributed over N processes, each process executes Sparse 1.3 for its set of points of the grid. The shaded areas are the ghost zones, i.e. the shared data area.

communications happen between horizontal and vertical neighbors; diagonal neighbors have less intensive communication (RODRIGUES *et al.*, 2009).

MPI assigns each domain to only one process, the only shared areas are the ghost zones. The less shared area, the less the external communication among MPI processes. The set of processes is distributed over the processors of the nodes, on our environment set we have a single node with four core processors.

Although BRAMS is in Fortran 90, it is possible to call C functions. However, the programmer must take some extra care with interoperability of both languages, in the next section we describe the main problems raised from this coupling.

3.1.1 Interoperability with Sparse1.3a

Sometimes it is necessary to mix different languages for a single executable; this usually happens when the programmer couples a library to its program. BRAMS coupled sparse1.3a, a C library with Fortran-ready interface. So no extra effort was necessary to adapt the C library.

To use this library with BRAMS, the programmer must enable Fortran interface. By enabling this option, Fortran programmers do not have to worry about interoperability problems, such as data type representation, arguments type (passed by reference or by value), and array representation (Row- or column-major order).

3.2 CCATT–BRAMS Codesign

Jacobi iterative solver has a parallel nature, which makes it suitable for FPGA parallelization (MORRIS; PRASANNA, 2005). The chemical kinetic process has diagonal matrices (ZHANG *et al.*, 2011), for our set of tests we guarantee they are strict diagonally dominant, this is necessary for Jacobi method convergence. Figure 17 shows a generic representation of BRAMS coupled to the hardware designed in OpenCL.

We implemented three versions of Jacobi method: Multi-threaded dense, Multi-threaded sparse, and single thread sparse. We describe them over the next subsections.

3.2.1 Jacobi Multi-threaded Dense

In our first attempt with Intel FPGA SDK, we programmed the FPGA using similar approaches used for GPU OpenCL. In other words, we used the concept of several computing units working in parallel to solve a problem in an SIMD mode. Although the source code is portable, performance is not, with this approach we support this affirmation.

We followed some of Intel best practices (INTEL, 2016a) for our first version, we avoided the divide operation for floating-point and used aligned memory in 64 bytes. This implementation

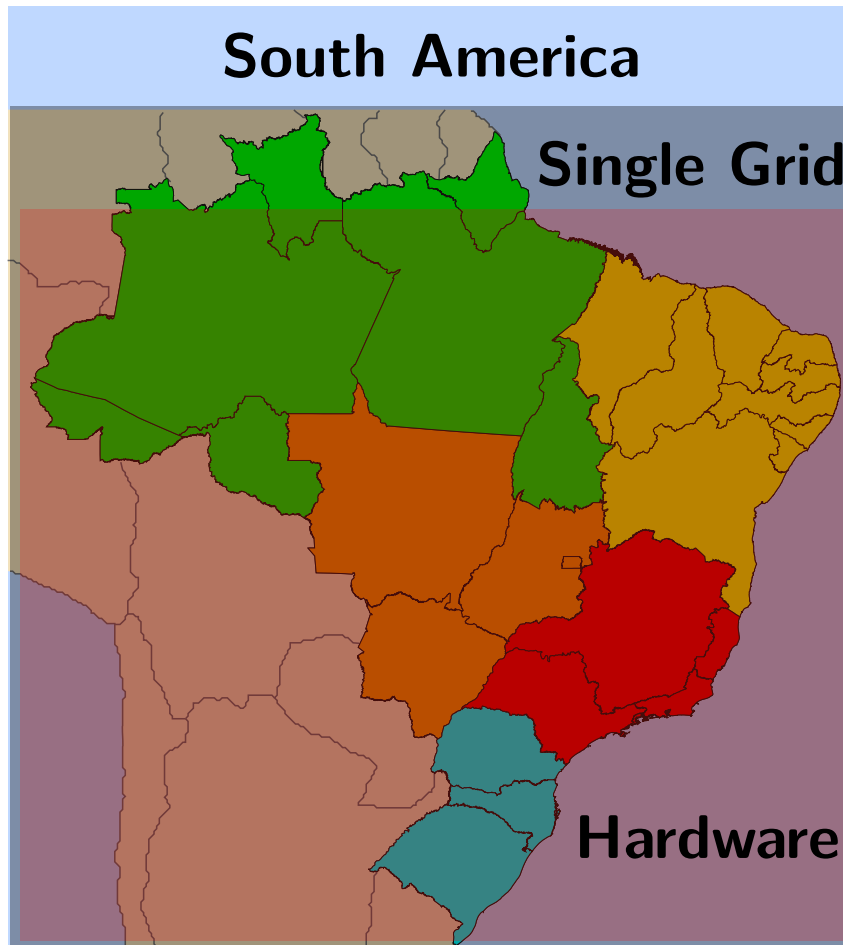


Figure 17 – Generic representation of BRAMS coupled to the Jacobi method in Hardware. In this figure, we present BRAMS over the South America with a single grid, red square represents Jacobi hardware circuit. Such circuit computes all the points over the grid.

uses a dense representation of matrix A . [Figure 18](#) shows the schematic of the first version of Jacobi Method.

Here we present m pipelines. Each pipeline can process different work-items from various work-groups. During kernel compilation, we can choose two options for work group size: **max_work_group_size** and **reqd_work_group_size**. The first option is a hint of the maximum number of work items; the compiler must not exceed this number.

The second option is much more strict, and it does not let the compiler optimize the work-group size for the problem. Work-group size is the number of work items for each work-group. We tested both options, and the first generated the best hardware. [Figure 19](#) depicts how work-groups and work-items map to our linear system.

According to the Verilog generated by Altera compiler, the compiler issues two work groups in hardware for a single compute unit, and all the remaining work groups are scheduled, i.e. two work-groups are executing at the same time. The option **num_compute_units** allows us to increase the number of hardware available, although Intel does not recommend to do so due to the concurrent access to global memory. Each hardware group has three stats:

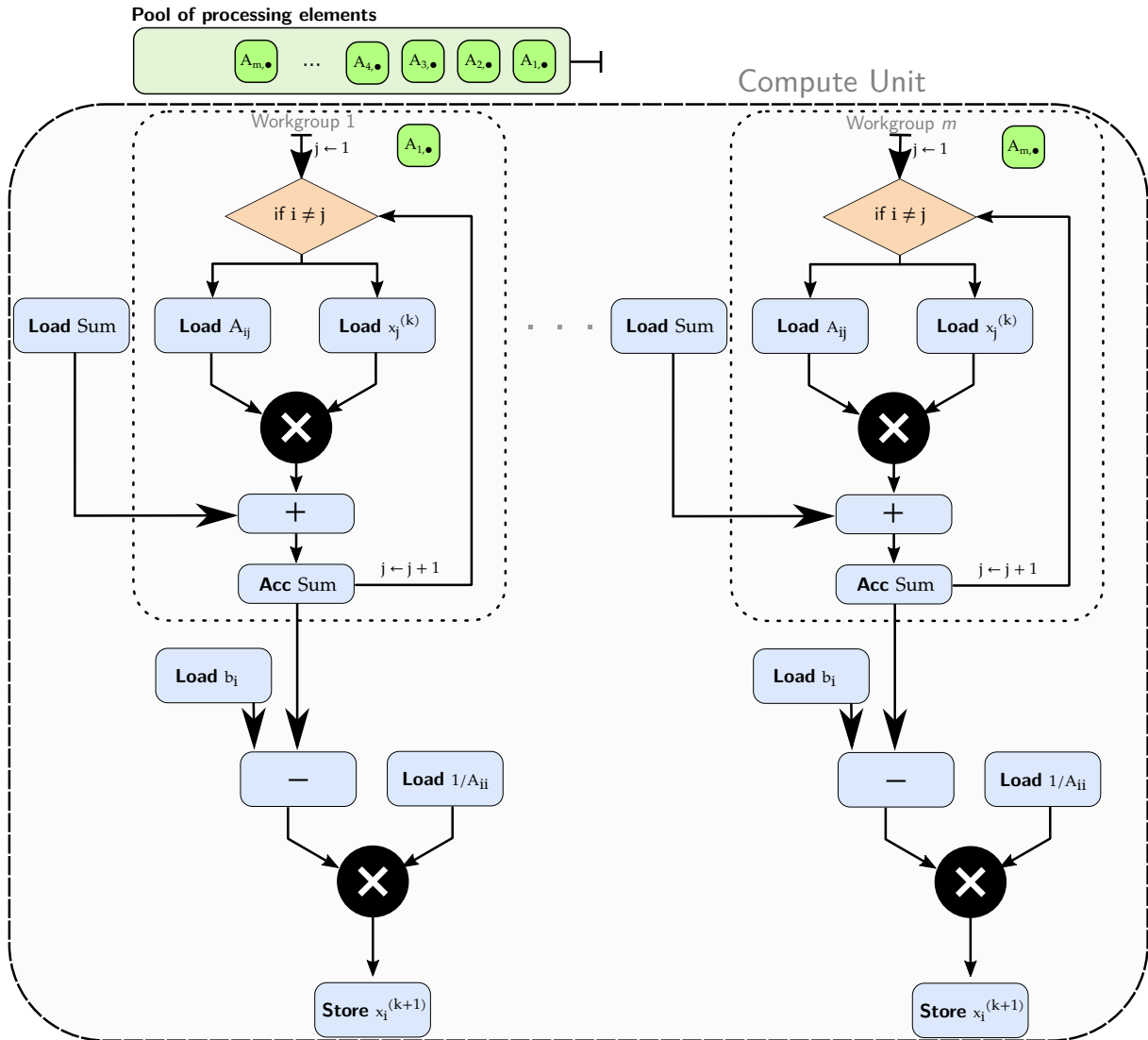


Figure 18 – Schematic of Jacobi method with multiple threads with dense representation.

1. NEW — Prepare to issue new work-group;
2. ISSUE — work items in a work-group;
3. WAIT — Check if ready to accept new work-group or if all done.

A better approach uses SIMD work item, which uses a single compute unit for performing in parallel an amount of work. The latter is only possible for work group size, whose number is a power of two. Our design comprises 47 work groups (processing element) with 47 work items each, i.e. our problems uses a prime number. We did not pad matrices with zeros to avoid transferring additional data.

After each work-group finishes, it stores the final result in the DDR3 memory (the global memory). The host can fetch data from the global memory of the device through PCIe interface.

The orange block represents the condition of Jacobi Method, where all non-diagonal elements of the matrix line must perform a multiply-accumulate (MAC) operation. Only then

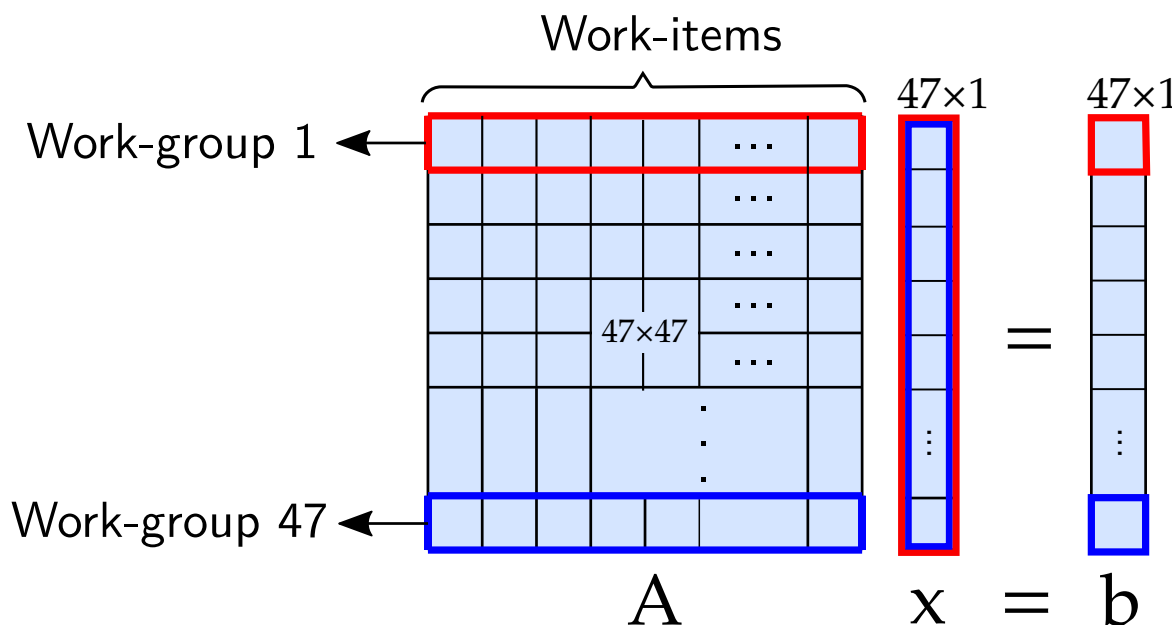


Figure 19 – Each color represents one work-group, each work-group has 47 work-items. According to the verilog, two work-groups are executing at the same time.

it is possible to load the Matrix A and x_{th} element from global memory for multiplication. We accumulate each result of the multiplication in a local memory.

Once traversed the entire line (green square), the hardware loads b vector element and performed a subtraction afterward. Then we perform a multiplication from this result with a load of $1/a_{ii}$; this is the final solution for an item of the result vector. As we have 47 work-groups and only two real work-groups, the final result is only available after the runtime schedules all the work-groups.

Each work group requires its memory space; this restriction does not allow OpenCL to synchronize the work-groups (MUNSHI, 2009, p. 27). That becomes visible in Jacobi hardware; not all work-groups are processing at the same time. This asynchronism forced us to calculate the vector norm in software.

To calculate the vector norm, the programmer must enqueue a read from the result buffer and execute it in software. Another possible solution is to have a specific kernel to calculate it, but this would require two kernel enqueues. For a huge vector, this is not a problem, since the execution time is higher than API delay time, which it is not our case. Jacobi method in Hardware and norm in software take turns until result converges to the desired solution with minor or no error.

3.2.2 Interoperability with Jacobi Multi-Threaded Dense

In this subsection, we explain how interoperability works between Fortran 90 and Jacobi method in FPGA, for such approach we used OpenCL. Each OpenCL application requires a host and a device code. The host is in C/C++, we design the FPGA circuit in OpenCL language, also

called OpenCL C programming language (MUNSHI *et al.*, 2011).

As we saw in Subsection 3.1.1, it is possible to call a C function from Fortran. However, now we had to take care with data type representation (double or Real*8); arguments from Fortran are passed by reference, in C they are passed by value; array representation in Fortran is column-major, in C they are row-major. Indices in Fortran start from 1, not 0.

In this version, we solved one linear system at a time. That is the same approach used by Sparse1.3a. In fact, we replaced the previous sparse function to ours.

For using our solver, we first needed to reprogram the FPGA on the system. We guarantee that with two similar approaches. The first requires the programmer to load the FPGA binary (AOCX) in advance through Intel SDK command, and set a variable:

```
1 $ aocl program <device> <file>.aocx
2 $ export CL_CONTEXT_COMPILER_MODE_ALTERA=3
```

The second reprograms the FPGA during runtime on the creation of program structure. It is user's decision to choose what is the best approach. Usually, the delay of reprogramming is about one or two seconds. Because of this overhead, we put all the kernel definitions in a single cl file.

After reprogramming, BRAMS execution starts and when it gets to chemical reactivity it starts solving the matrices through FPGA circuit designed in OpenCL. Apart from creating the necessary OpenCL structures, we use C language as a bridge between Fortran and FPGA circuit. Figure 20 shows how BRAMS calls the circuit. Different from the GPU, our cl file represents a physical circuit in FPGA not software.

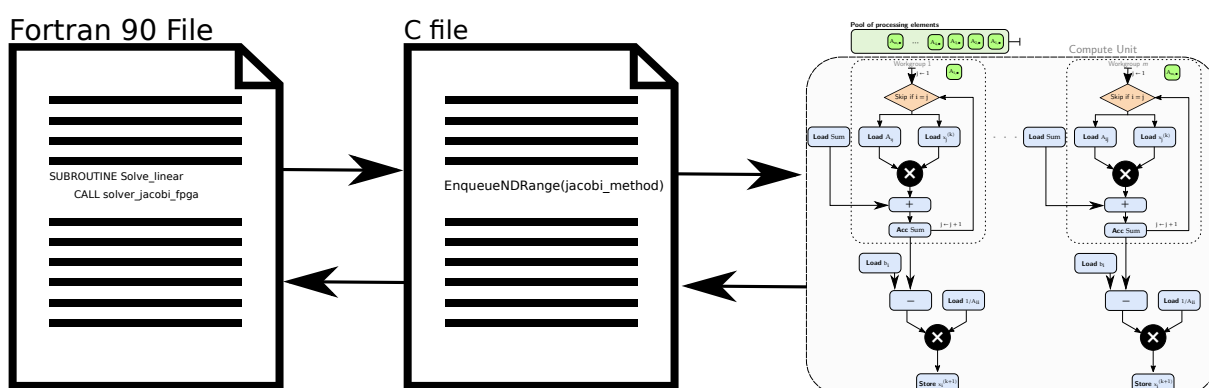


Figure 20 – Interoperability of BRAMS and OpenCL. Fortran calls a function from the C host, which in turn is responsible to manage the device.

As we can see, software and hardware have a strong communication. The C file is on the host processor, and it is responsible for calculating the vector norm for each iteration of Jacobi. If the result did not converge, we send all the necessary data and NDRange. A host that uses NDRange bigger than 1 expects a circuit with data parallelism.

Any communication between FPGA and host requires 64 bytes aligned memory; not doing so degrades transfer time. In Fortran, we could not align `spack2d` struct, because the directives for aligning memory in Fortran does not work for derived types (KRISHNAIYER, 2015).

For avoiding transfer time penalty, we had to copy the matrix into an aligned vector in C. We considered the modification of `spack2d` struct, but that would require a significant adjustment on BRAMS source code, which could lead to undefined behavior and results.

The numbers of communications equal the number of Jacobi iterations; We set 150 as a maximum number of iterations. We used previous experience to define this threshold.

3.2.3 Jacobi Multi-threaded Sparse

During the first approach, we realized that much of the execution time was in I/O. For each iteration we send an enormous amount of zeros, to improve the I/O time we changed the matrix representation from dense to [Compressed Row Storage \(CSR\)](#).

According to [Fernandes \(2014\)](#), around 10% of the matrix elements are non-zeros. So in our first attempt, we defined the NNZ to be 10% of 47×47 . During execution time, we received segmentation fault due to the lack of memory to accommodate the NNZ.

By analyzing the NNZ of matrices, we noticed that this number ranges from 8% up to 25%. Allocating and freeing device buffer during runtime is very expensive; we must choose a fixed size. At this point, we reached a trade-off: a small NNZ is risky, and a big NNZ sends additional data to the device. We chose the later by defining the buffer size according to the [Equation \(3.1\)](#), CSR format saves memory, whose NNZ is not bigger than this equation.

$$\frac{N(M-1)-1}{2} - 1, \quad (3.1)$$

where: N = Number of rows

M = Number of columns.

Choosing NNZ size according to [Equation \(3.1\)](#) does not improve I/O because we are not saving any space; we just avoid filling the pipeline with zeros. This new architecture in [Figure 21](#) is very similar to [Figure 18](#), but now we avoid additional calculation by using CSR format. In this version, we still must perform vector norm computation on the host, because of the same reasons of the dense version.

The main difference in this architecture is the internal loop (dotted square); we only read the NNZ values from the i^{th} row. In loads we use CSR representation, note that we use lower case for all the loads, this is because CSR representation uses three vectors. Since each line has

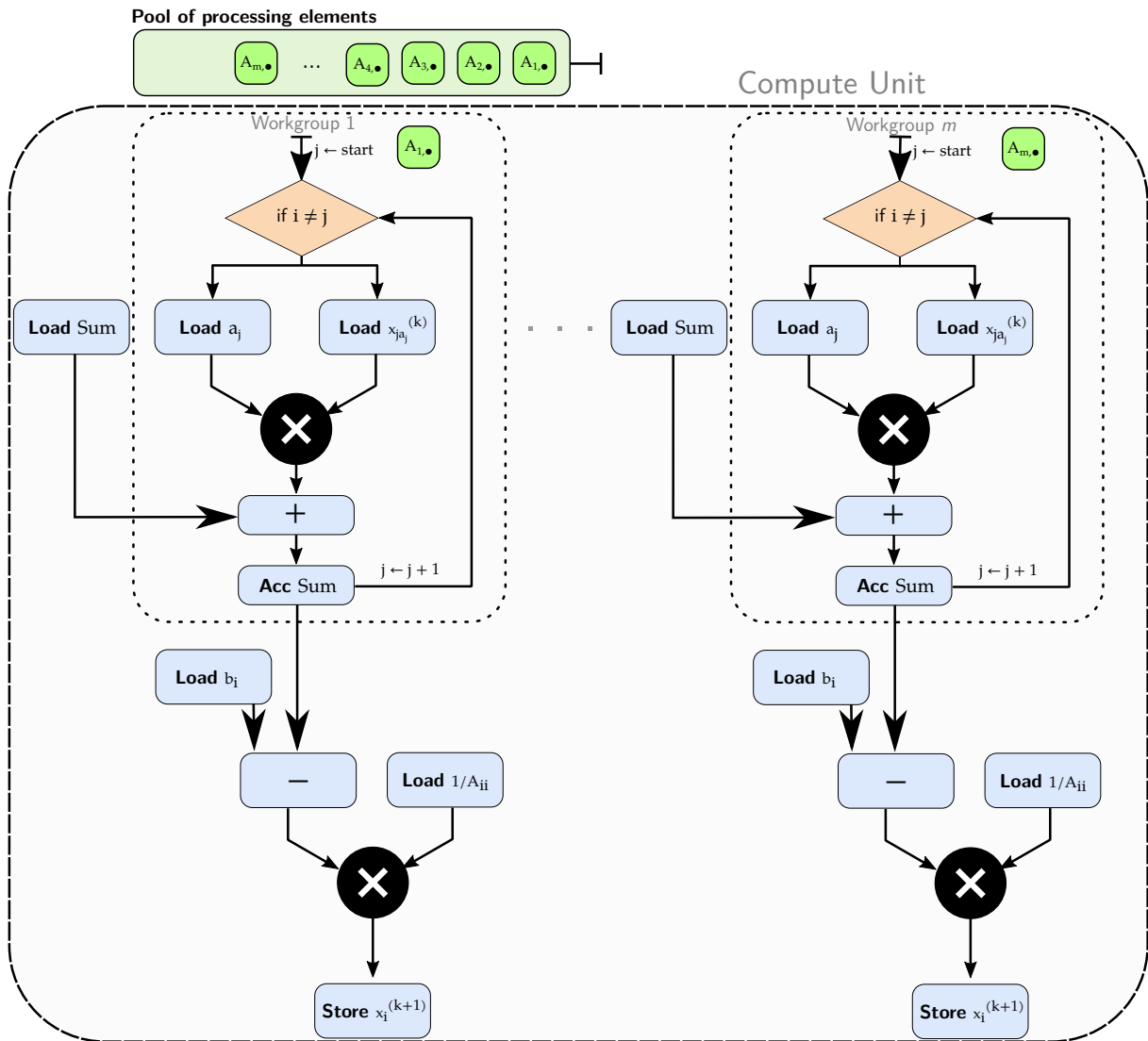


Figure 21 – Schematic of Jacobi method with multiple threads with sparse representation.

its NNZ size, we defined each group to have one work item to process each line; we show this scheme in Figure 22.

3.2.4 Interoperability with Jacobi Multi-Threaded Sparse

Changing the matrix representation required few modifications on host code. We added a C algorithm for converting from dense representation to CSR; we called `dnscsr_c`, this function relies on the `dnscsr` algorithm from Sparsekit¹.

Sparsekit `dnscsr` function is in Fortran 90, using this library does not require any modification to BRAMS source code. We decided to rewrite this function in C to avoid managing different source codes.

Refactoring the algorithm added the burden of managing row- and column-major order

¹ <https://people.sc.fsu.edu/~jburkardt/f_src/sparsekit/sparsekit.html>

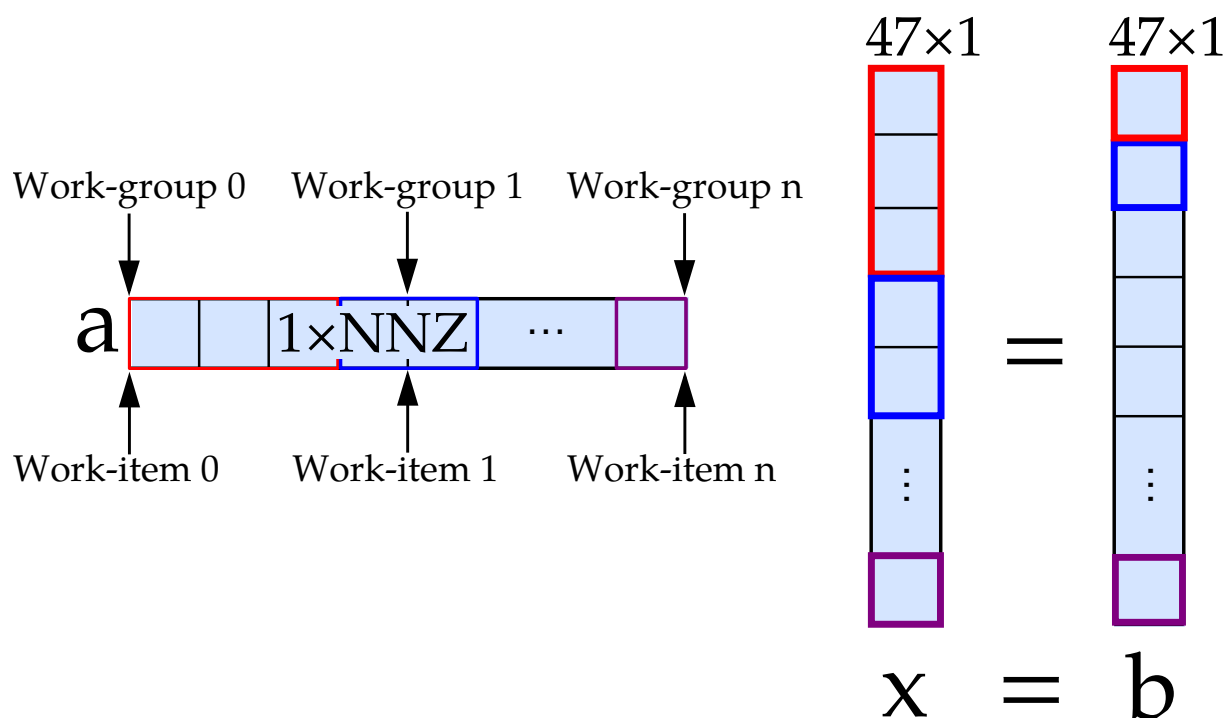


Figure 22 – Each color represents one work-group, each work-group has one work-item. In this manner, the number of work-items is equal to the number of work-groups.

of matrix A. So in Fortran we passed the transposed matrix of A to `dnscsr_c`; we used transpose for simplicity of the algorithm in C, and avoid undefined behavior. This is not a problem, since we are not using dense representation.

As we told in [Subsection 3.2.3](#), the NNZ varies from one matrix to another. So it is not possible to determine a single size of the vectors Value and JA. Allocating and freeing memory for each matrix is not a good option either, since this increases execution time. So, in our approach, we declare three vectors Value, IA, and JA, where Value and JA have fixed size according to [Equation \(3.1\)](#).

The host code of OpenCL uses these vectors that come from Fortran. Thus, we avoid creating them for each matrix during execution. Fortran is also responsible for calling `dnscsr_c` before any call to Jacobi Multi-threaded Sparse solver.

3.2.5 Jacobi Single-threaded Sparse

According to [Intel \(2016a\)](#), sharing fine-grained data is not suitable for data parallel programming model. In these cases, using a single thread kernel can offer a higher throughput.

In the two previous architectures we could not compute vector norm, due to the restriction of different memory space for each work group; and consequently, we are not able to share memory.

In a single thread; we can compute vector norm because in this case, OpenCL works like a C to Hardware. We can also avoid dispatching multiple times the same data from CPU to

FPGA since we can define in our kernel to execute Jacobi until the matrix converges.

For using single thread kernel, we must set `reqd_work_group_size (1,1,1)` and not use `get_global_id`. When designing the hardware as a single thread, the programmer must declare the variable in the deepest scope possible; otherwise, it will consume more hardware resource than it needs. AOCL (Altera OpenCL) generates a detailed report containing the depth of the variables; we improved the depth of our variables through this report.

Regarding loop pipelining, single kernel executes multiple iterations in flight. Not all loops are capable of pipelining; the outermost loop (plain square), responsible for controlling the iterations of the algorithm, is an example. The algorithm demands the loop to stop when it converges since each matrix have different stop criteria the compiler cannot infer any order.

CSR format adds a complex loop exit condition in the pipeline (dotted red square) since each row has different NNZ; Note the red border square, it means that it is necessary two loads. Another problem is the loop-carried dependency on the sum accumulator. Such condition, multiple loads, and loop-carried dependency degrade loop pipeline performance because subsequent iterations cannot launch until the previous one completes. So for Jacobi single sparse, dot product is the bottleneck.

During vector norm calculation (innermost dotted black square) we also have a loop-carried dependency on the accumulator, but now there is a well-formed loop. That is, the loop is always the same independent of iteration. Such loop allows us to remove it by inferring shift registers. So the vector norm calculation becomes parallel. In [Figure 23](#), we show the pipelines for Jacobi.

Besides the different architecture, this kernel is 100% in hardware. So we do not avoid any expensive operation; we used built-in functions for power and square root calculation. This architecture represents the Jacobi method algorithm in hardware, totally independent from the host processor.

[Figure 23](#) shows a single compute unit. Although the compiler allows the programmer to infer more compute units in a single kernel, the behavior does not make sense. Define more than one compute unit for a single thread kernel means the same calculation over different circuits. Different from NDRange, the amount of work is not divisible.

Each matrix belongs to a single kernel; the programmer can increase throughput by replicating the kernels manually and calling them with different queues and matrices on the host. As already stated in the previous architectures, the programmer must consider if this a good approach due to memory competition.

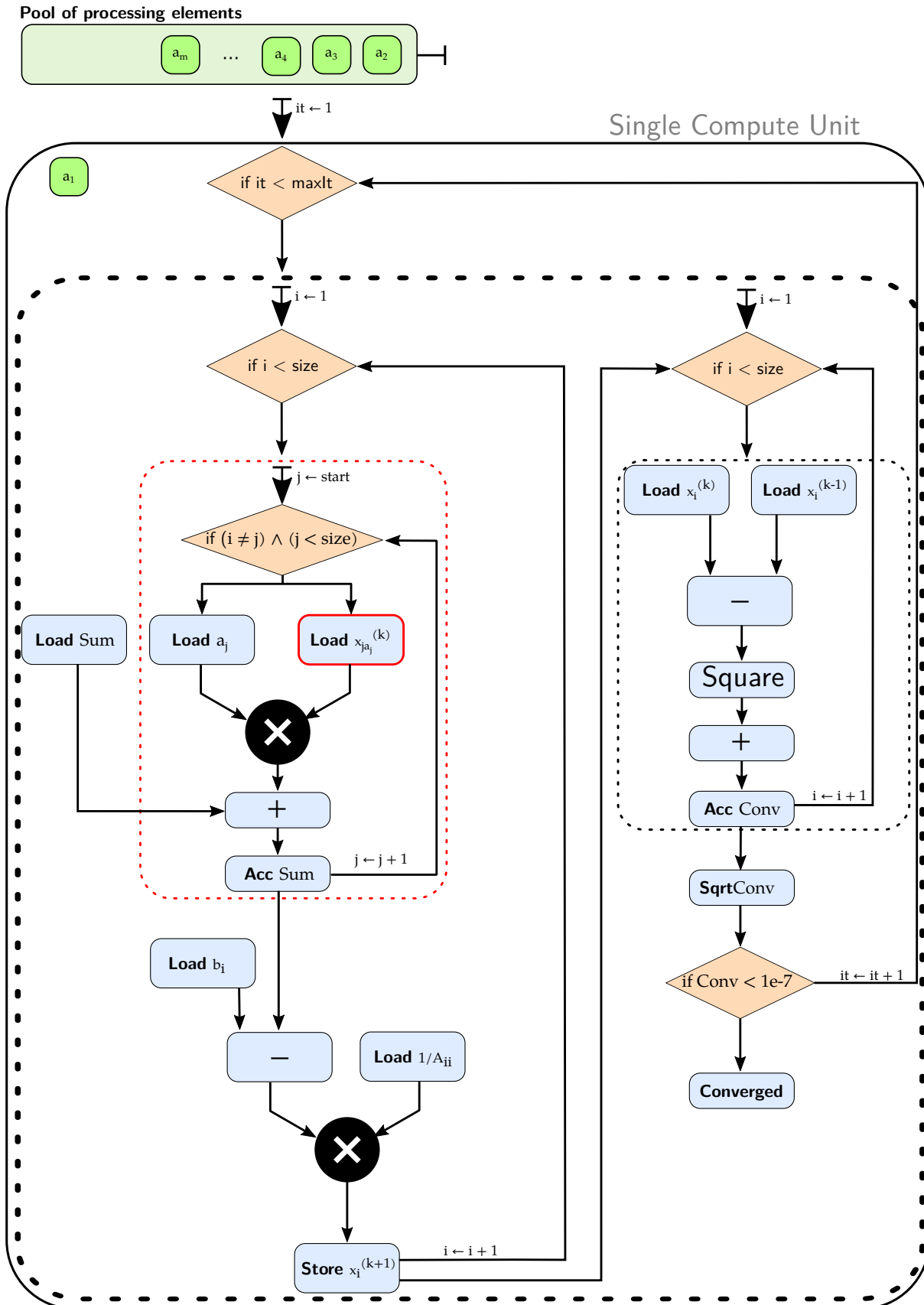


Figure 23 – Schematic of Sparse Jacobi method with single thread.

3.2.6 Interoperability with Jacobi Single-threaded Sparse

The complex hardware architecture allowed us to improve host code because we do not have to keep track of convergence. It also improved I/O exponentially, now we send the data only once, and receive the result back.

Regarding I/O we implemented two modifications. First, we changed the buffer allocations; we noticed that we could boost performance by using pinned memory. This kind of memory resides in a non-pageable space, by using such allocation we avoid one extra copy from pageable memory to non-pageable; we show this in [Figure 24](#). It is important to point out that a huge amount of pinned memory can degrade system host performance by decreasing pageable memory space ([CHENG; GROSSMAN; MCKERCHER, 2014](#)).

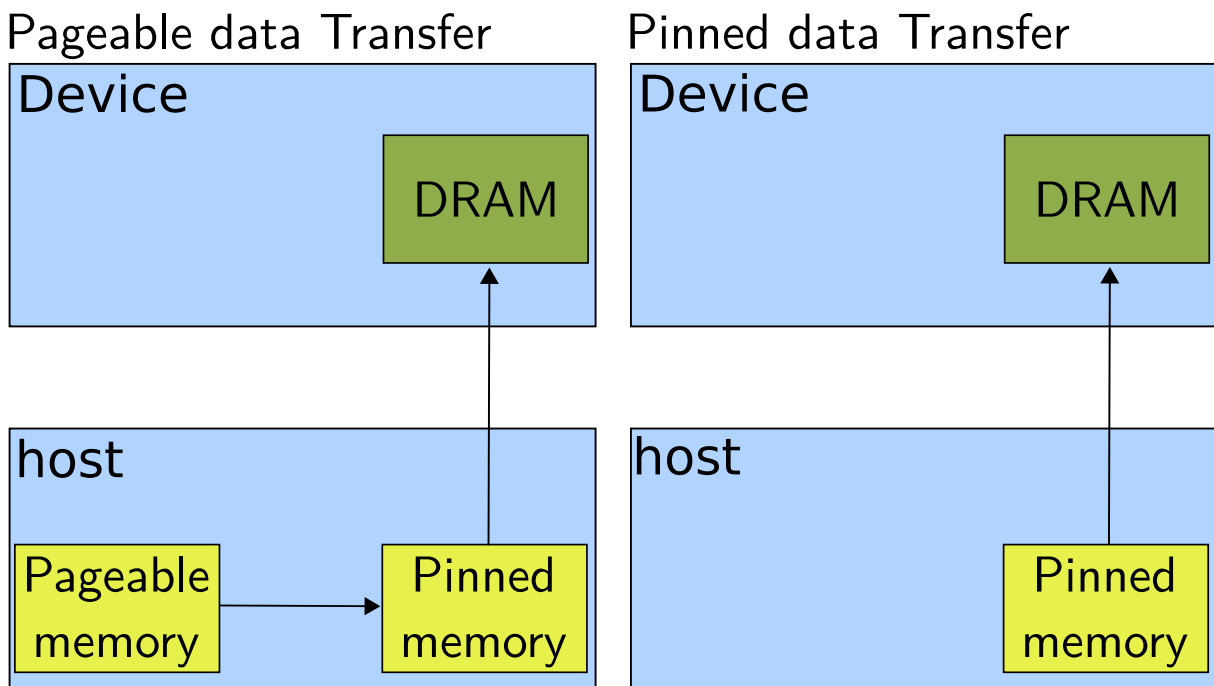


Figure 24 – How pageable and pinned memory data transfer work, based on [Harris \(2012\)](#).

In the second modification, we implemented a double buffering scheme. This method hides I/O latency, while the pipeline is processing one matrix, another one is being transferred from the host to FPGA. Double buffering is kernel independent, i.e. the programmer order the enqueues to overlap communication and computation. Since Intel FPGA SDK does not implement out-of-order queues, we had to use two queues for using double buffering.

With exactly same function, we could call this new hardware from Fortran. The biggest change was on the underlying hardware. Now we launch the kernel as a task, not as NDRange. So, for each call from Fortran, we have a task parallelism.

RESULTS

In this chapter, we present the profiling from BRAMS. In this profiling we confirm that the chem term is the most expensive equation in the mass continuity equation. Then we show the experiments with the hardware in FPGA to solve the linear system from this equation.

We perform three experiments, one for each architecture of Jacobi method, we also present the results from Sparse1.3a. We used the same heterogeneous machine we used for profiling BRAMS. Regarding the analysis, we discuss precision, time execution, and energy.

4.1 Result analysis

We used Gprof to profile BRAMS under two configurations: with the chemical module enabled, and with the chemical module disabled. Such configuration is possible through the RAMSIN file, the input configuration file. We discuss the results below.

In the first configuration, we realized that the function *radrrtm_* is responsible for 68% of the total time of computation. That is the only function that exceeds 3% of the total time.

On Figure 25 we can see the call graph, we generated this graphic representation with *gprof2dot*. It is important to point out that this call graph is not complete, we defined some thresholds to make the call graph fit the page. For this discussion, this call graph is enough.

When we enable the chemical module, we note that *rodas-3* from Rosenbrock method takes about 41% of the total time. We represented the call graph on Figure 26; again this is a partial call graph.

The second heaviest function is *spFactor*; it is responsible for 17% of the total time. This function decomposes the matrix in LU format, as we mentioned earlier this decomposition is $\mathcal{O}(n^3)$. By comparing the call graph in Figure 25 with Figure 26, we noted that solving linear systems in *rodas-3* are two times more expensive than computing radiation.

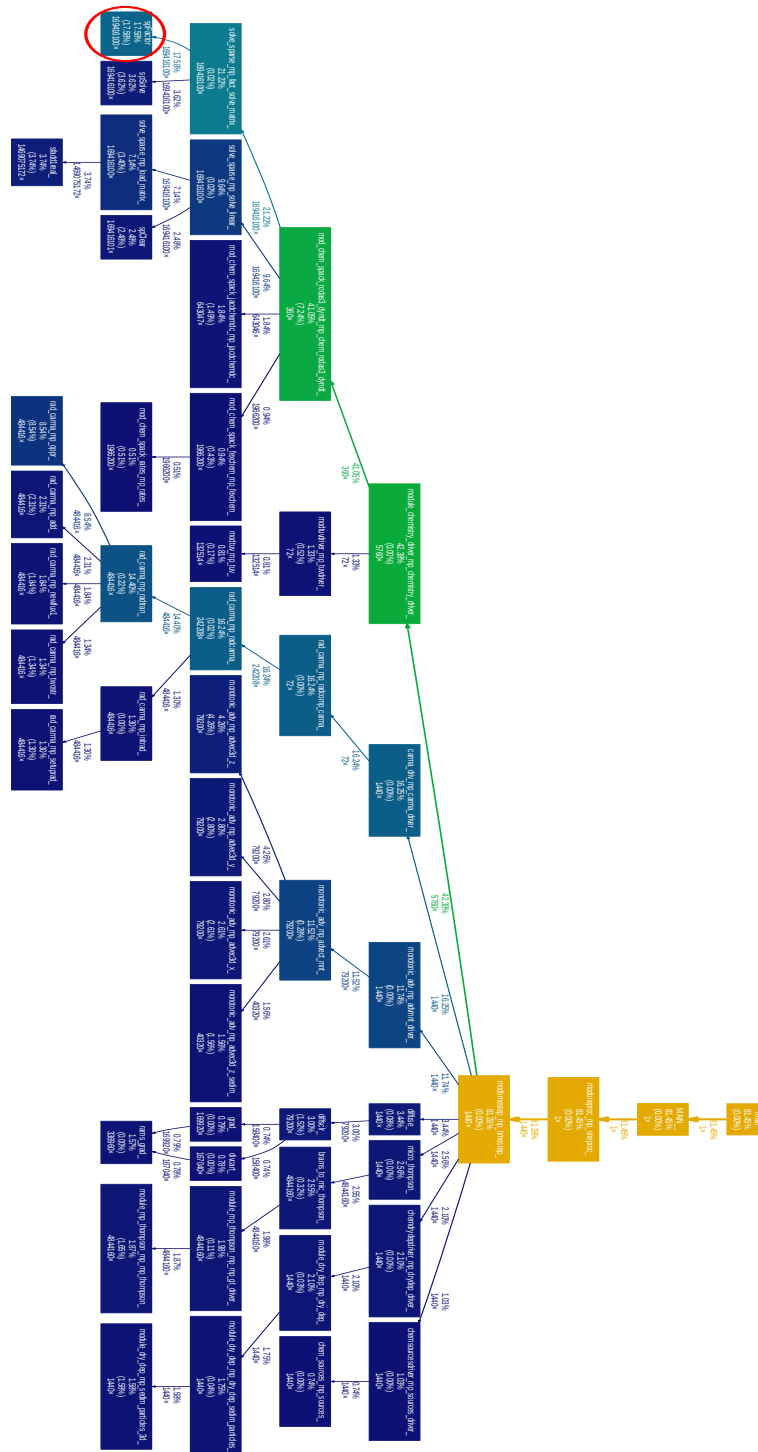


Figure 26 – Call Graph for BRAMS with chemical module enabled.

single MPI process.

According to HRZ (2017), the fine-grained scheduling available in GPU is not present on the FPGA circuit generated by Intel OpenCL SDK, i.e. two processes cannot share the FPGA circuit. Even with this information, we decided to perform some tests to reinforce this affirmation.

In our first attempt, we replicated the kernel for each MPI process. That requires the knowledge of how many processes will access the device simultaneously. Figure 27 shows the

schematic for our explanation using two processes.

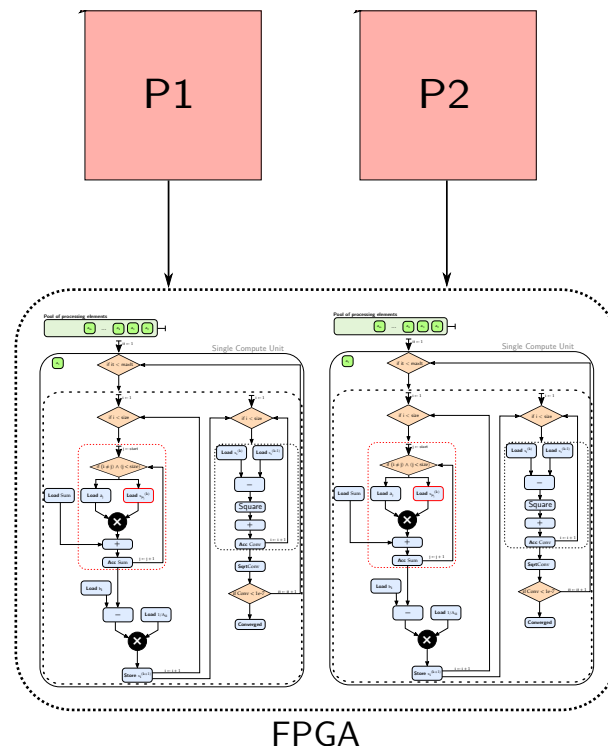


Figure 27 – Each MPI process accesses a copy of the kernel (an FPGA circuit).

For using this scheme, FPGA reconfiguration must occur before BRAMS execution. Otherwise, both processes try to reconfigure the same FPGA and a deadlock occurs. During execution, we realized that this approach is not feasible; the second MPI process is not able to continue execution because Intel FPGA SDK kills it.

We tried another method; we decided to share OpenCL structures by using inter-process shared memory in Linux¹. Figure 28 shows that each process has a private Context and Queue, we share program, device, and memory. We can see the program as a set of FPGA circuits, where each circuit represents a kernel. The device represents the FPGA board; each process declares a memory region inside the FPGA (device), in the figure we represent them as two purple boxes.

The first problem arose from this approach was the execution order of the MPI processes. MPI does not impose any order in its pool of processes; we needed to execute the master process before any other process. The master process was responsible for creating the shared region. During our tests, with two processes, we could infer an order randomly.

Considering the cases where there was order among processes, we still reached the Intel runtime restriction. Regarding the out-of-order cases, a segmentation fault occurred because there was no program structure.

We tried to avoid this problem by using a mixed approach. We programmed the FPGA before BRAMS execution, as we tried in the first approach, and execute BRAMS with shared

¹ http://man7.org/linux/man-pages/man7/shm_overview.7.html

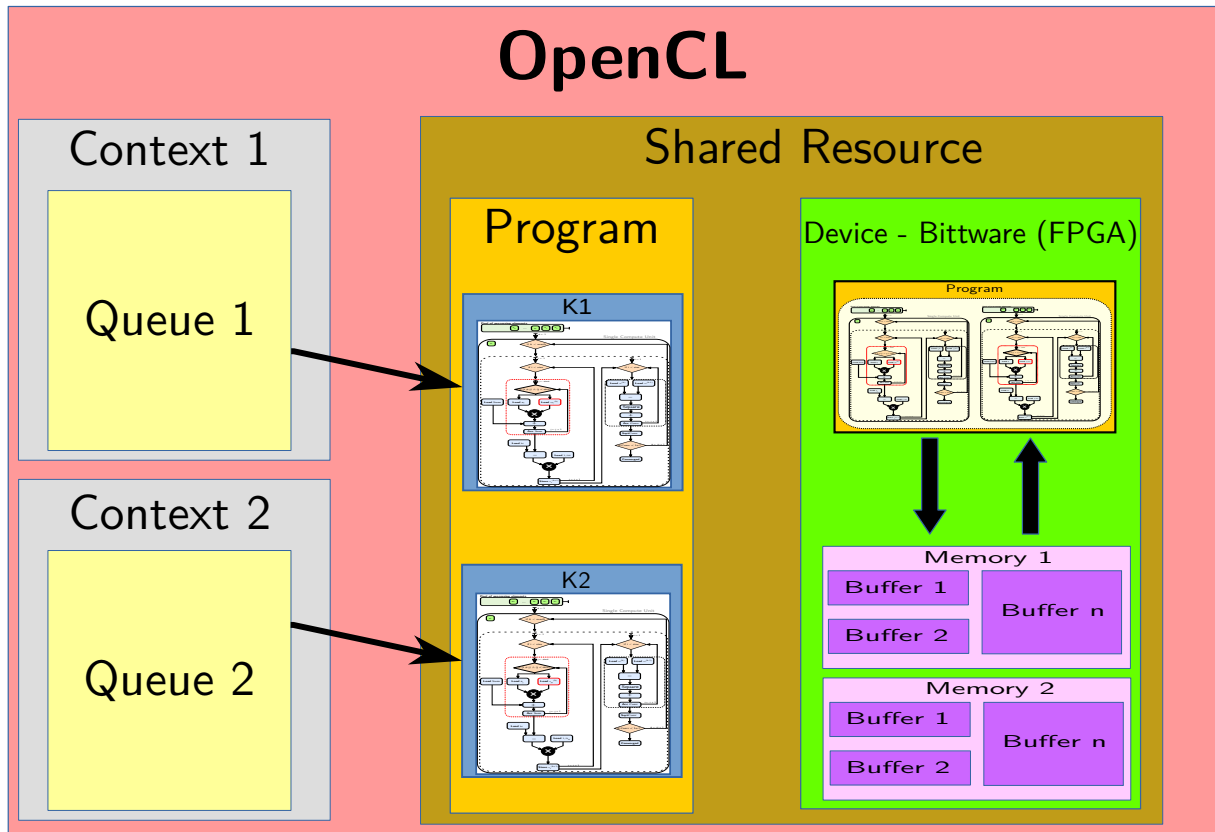


Figure 28 – OpenCL data structures – Program, Device and Memory buffers are shared among MPI processes.

memory. This approach did not require to make any modification to BRAMS source code. Although we avoid the order problem, runtime still kills the second process. These tests could confirm that each device requires a single process access.

4.3 Results from Jacobi Multi-threaded Dense

In our first design, we programmed our FPGA as GPU, i.e. we used the concept of several working units working parallel. Despite the difficulty to understand, this programming model is very common for parallel programming, and we found several tutorials teaching how to program simple problems in OpenCL. Jacobi method is similar to matrix-vector multiplication, a general operation in OpenCL programming.

During this approach, we realized the host is as important as the kernel. In this attempt, we committed several errors that led to a severe degradation in performance. We will discuss the kernels problems, and then host problems.

The memory hierarchy is critical regarding kernel programming. We have three memory types: Global, Local and Private memory. We tried to use the general practice of avoiding memory access to global memory by copying the contents to a local memory.

Our designs usually had a higher frequency when we avoided this practice, after some

digging on the source code behavior we discovered that compiler optimizes accesses to constant global memory. When the programmer defines the global memory as a constant, which is the case for the matrix A and vector b in the linear system, the compiler creates a cache in on-chip memory to store these arguments, and it allows to cache the data for each load (INTEL, 2016a).

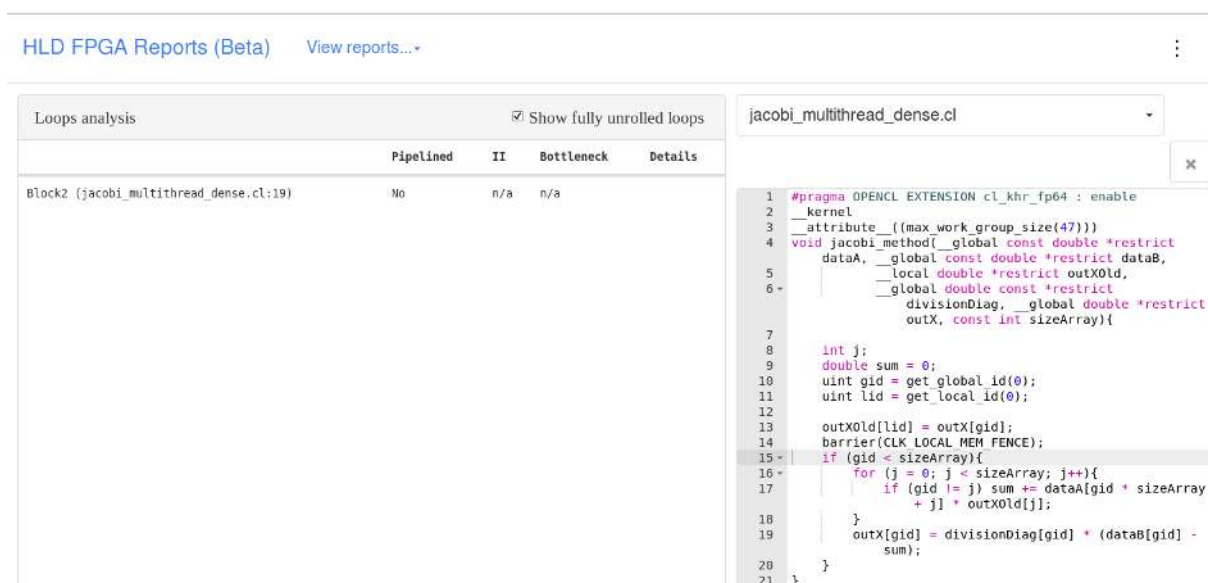
Intel FPGA SDK allows the programmer to define two types of cache. The first approach requires a key `__constant` for all constant parameters; in this manner, a single cache is responsible for storing all the values from different constant parameters. The default cache size is 16Kb; however, the programmer can define another size during compilation. Using a single constant cache did not improve the performance of our kernel, in our case, this approach decreased the kernel frequency.

A second method uses `__global const`, which creates a cache for every parameter, i.e. no shared cache. According to our results, this configuration led to higher frequency kernels.

Copying from global memory to local memory is useful when data is not constant, otherwise it is not worth it. We used this approach for x in Jacobi method; this copy requires a barrier to guarantee that all work-groups have a copy of x . Each iteration of Jacobi updates the current solution and uses it for the next iteration.

It is important to point out that the pipeline for NDRange in this architecture is not the same pipeline for the single thread. This pipeline accepts multiple work-items from different work-groups; thus there is no pipeline for loops in the kernel.

This pipeline differences became apparent with the graphical reports from Intel FPGA SDK. Although the user may compile the project on a computer with Ubuntu (operational system), the tool requires Red Hat or CentOS to generate this report. Figure 29 shows the report, regarding loop pipelining, for this architecture.



The screenshot displays the Intel FPGA Reports (Beta) interface. On the left, a table titled "Loops analysis" shows the results for a kernel named "jacobi_multithread_dense.cl". The table has columns for "Pipelined", "II", and "Bottleneck". The entry for "Block2 (jacobi_multithread_dense.cl:19)" shows it is not pipelined, with "II" and "Bottleneck" values of "n/a".

Loops analysis	Pipelined	II	Bottleneck	Details
Block2 (jacobi_multithread_dense.cl:19)	No	n/a	n/a	

On the right, the source code for "jacobi_multithread_dense.cl" is shown. The code includes a pragma to enable the OpenCL extension for kernel pipelining. The kernel function "jacobi_method" takes several arguments, including a global constant "divisionDiag". The code implements a nested loop structure where the inner loop calculates a sum and updates the output array "outXold".

```

1 #pragma OPENCL EXTENSION cl_khr_fp64 : enable
2 kernel
3 __attribute__((max_work_group_size(47)))
4 void jacobi_method(__global const double *restrict
5 dataA, __global const double *restrict dataB,
6 __local double *restrict outXold,
7 __global double const *restrict
8 divisionDiag, __global double *restrict
9 outX, const int sizeArray){
10
11 int j;
12 double sum = 0;
13 uint gid = get_global_id(0);
14 uint lid = get_local_id(0);
15
16 outXold[lid] = outX[gid];
17 barrier(CLK_LOCAL_MEM_FENCE);
18 if (gid < sizeArray){
19     for (j = 0; j < sizeArray; j++){
20         if (gid != j) sum += dataA[gid + sizeArray
21             + j] * outXold[j];
22     }
23     outX[gid] = divisionDiag[gid] * (dataB[gid] -
24         sum);
25 }
26 }

```

Figure 29 – Pipeline report for Jacobi multi-threaded dense.

This report describes the information about the developed kernel. On the left of the figure is the loop analysis; in this field, we have four tabs. The first tab indicates if it pipelined the loop in line 19; the compiler does not support loop pipelining in NDRange. II represents how many cycles this loop must wait before processing the next iteration; as we do not have loop pipelining, this value is n/a.

Regarding the third and fourth columns, as the name says, they are related to the bottleneck of the pipeline and the details of how the compiler optimizes the loop. Programming like a NDRange does not generate a report with much information. On the right, we have the source code of the kernel. The report highlights which line is consuming more resources.

Although we could improve kernel area utilization and performance with the reports, communication CPU-FPGA the was a problem. For every iteration of Jacobi, we needed to exchange information with the host, because we could not perform vector norm in hardware. [Figure 30](#) shows this strong communication.

This communication exchange caused a severe drop of performance in BRAMS. In average, for each matrix computation, we needed 28ms. Intel provides a profiling tool that is capable of measuring the communication and kernel computation time accurately. We show the communication and execution time in [Figure 31](#).

In the kernel execution tab, the tool presents the kernel execution time (jacobi_method) and memory transfers. Note that the execution starts when the FPGA receives the first chunk of data; the communication is dominant in time execution.

Besides the time execution, we wanted to check the occupancy and stall of the kernel pipeline. This first measures the amount of work we perform; the second measures the contention in the memory bandwidth.

According to [Figure 32](#), we do not have enough work for our kernel; our highest occupancy is 52%. Considering that we are filling the pipeline with zeros, the occupancy for actual computation must be even smaller. Regarding the stalls, we have a worst-case scenario of 9.29%; the ideal percentage is zero, i.e. almost no contention in the memory.

Lastly, we present the statistics for this architecture in [Figure 33](#). As we can see, our design still allows much improvement regarding memory bandwidth; this requires significant change on BRAMS structure of the chemical reactivity.

From our experience with the tool, we do not recommend to execute kernel profiling for more than a few seconds to BRAMS execution. We tried to generate a final profiling with the entire execution, and it generated a file with more than 25GB. Reading such file was not possible due to memory restrictions imposed by our machine. We believe that this behavior must be similar for other huge applications. Although Altera ensures that their profiling hardware counters do not increase any overhead, we had different results in this dissertation and we did notice some overhead.

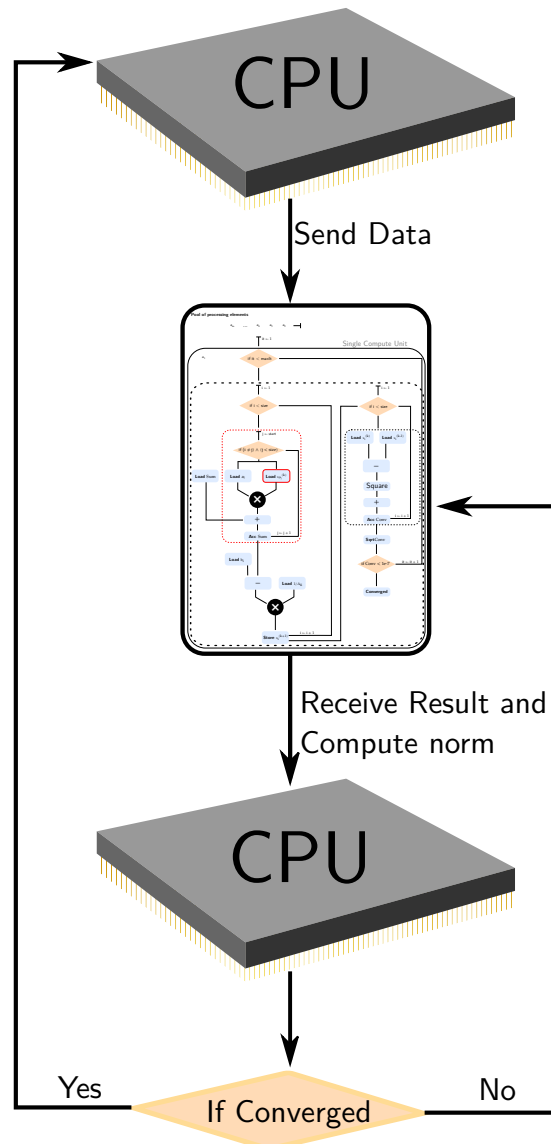


Figure 30 – CPU communicates with FPGA for every iteration. CPU sends to the FPGA the initial data, after FPGA processing it, the FPGA returns the result to the CPU, which in turn computes the vector norm and decides if it sends another data or computes another iteration.

Despite the unsatisfactory performance of this architecture, we wanted to prove that our design was correct. We measured the error in the Rosenbrock Method, this method computes a linear system, whose solution depends on Jacobi Method, and updates vector b for the next stage. After the fourth stage, the algorithm computes the error and rounding of the block. If rounding is over the tolerance, Rosenbrock updates each matrix A of the block. In this manner, a new set of linear systems is available for each Rosenbrock stage.

Computing a 24-hour weather forecasting is not possible with this architecture due to time limitation. According to our extrapolation, it would be necessary 44 days to process this resolution. So we measured just the first block of error of the Rosenbrock Method; we compare the result with the first block of error in software using Sparse1.3a. We obtained a satisfactory error of $1.241371e - 19$.

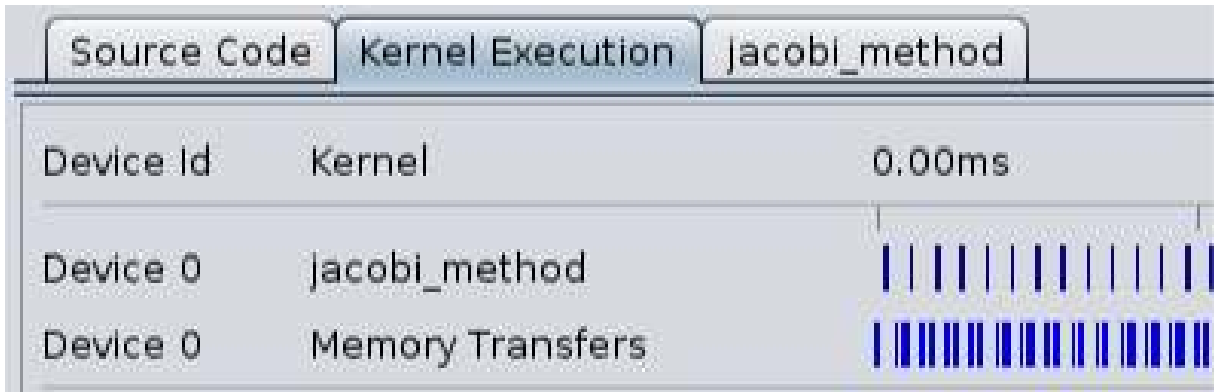


Figure 31 – Communication and execution time with Intel FPGA SDK profiling for kernels. Note that there is much more communication than computation.

15	outXOld[lid] = outX[gid];	0: __global{DDR}.read	0: 0.5%	0: 26.8%	0: 654.9MB/s, 99.69%Efficiency
16	barrier(CLK_LOCAL_MEM_FENCE);				
17	//#pragma unroll				
18	if (gid < sizeArray){				
19	for (j = 0; j < sizeArray; j++){				
20	if (gid != j) sum += dataA[gid * sizeArray + j] * outXOld[j];	1: __global{DDR}.read	1: 9.29%	1: 52.9%	1: 5111.5MB/s, 12.50%Efficiency
21	}				
22	outX[gid] = divisionDiag[gid] * (dataB[gid] - sum);	0: __global{DDR}.read	0: 0.1%	0: 26.8%	0: 17.0MB/s, 81.76%Efficiency

Figure 32 – Efficiency of Jacobi multi-threaded dense with Intel FPGA SDK profiling for kernels. The red line points that the global memory reads (line 22) are the bottleneck of the application.

Statistic	Measured	Optimal
Worse Case Stall (__global) %	9.29%	0%
Kernel Clock Frequency	305.2 MHz	na
Global BW (DDR:bank1 read-only)	3015.2 MB/s	12800 MB/s
Average Read Burst	1	16
Global BW (DDR:bank2)	2800.9 MB/s	12800 MB/s
Average Write Burst	3	16
Average Read Burst	1	16

Figure 33 – Statistics of Jacobi multi-threaded dense with Intel FPGA SDK profiling for kernels.

Regarding energy consumption, we used PowerPlay Power Analyzer Tool from Quartus II; this tool is responsible for estimating the potency in mW of the circuit. According to the tool, our design consumes about 11W. We summarize the results of this architecture in [Table 6](#) and [Table 7](#). In the latter, we split the execution time from transfer time.

Table 6 – Results from Arch 1.

Area	Frequency	Time	Energy	Error
19%	305 Mhz	~44 days	11 W	1.241e-19

Table 7 – Timing results from Arch 1.

CPU-FPGA	Execution	FPGA-CPU	Total Time
11686us	9153us	7806us	28645us

4.4 Results from Jacobi Multi-threaded Sparse

In the profiling of the previous execution, we noted that communication was not the only problem. We realized that there was extra computational time in the kernel; as we were using a dense representation, we chose to avoid filling the pipeline with zeros.

We decided to use CSR, a sparse representation suitable for [Sparse Matrix-Vector multiplication \(SpMV\)](#) (BELL; GARLAND, 2008). As we mentioned earlier, Jacobi method is very similar to the matrix-vector algorithm; we used this characteristic in the choice of CSR format.

In [Figure 34](#), we show the statistics related to stall, occupancy, and bandwidth. We compare each result with the previous architecture, which is the most similar architecture. We do not present the loop pipeline report for this architecture (NDRange does not pipeline the loops).

12	start = ia[gid];	(__global{DDR},read)	(2.94%)	(47.6%)	(598.3MB/s, 100.00%Efficiency)
13	end = ia[gid + 1];				
14	outXOld[lid] = outX[gid];	0: __global{DDR},read	0: 1.24%	0: 47.6%	0: 1174.1MB/s, 99.68%Efficiency
15	barrier(CLK_LOCAL_MEM_FENCE);				
16					
17	if (gid < 47){				
18	for (k = start; k < end; k++){				
19	if (ja[k] != gid){	(__global{DDR},read)	(1.78%)	(52.0%)	(317.3MB/s, 21.21%Efficiency)
20	sum += dataA[k] * outXOld[ja[k]];	1: __global{DDR},read	1: 3.29%	1: 52.0%	1: 408.6MB/s, 26.85%Efficiency
21	}				
22	}				
23	outX[gid] = divisionDiag[gid] * (dataB[gid] - sum);	0: __global{DDR},read	0: 2.09%	0: 47.6%	0: 127.4MB/s, 19.62%Efficiency

Figure 34 – Efficiency of Jacobi multi-threaded sparse with Intel FPGA SDK profiling for kernels. Sparse format causes a severe drop of performance when saving the results back to the global memory.

The red rows represent the worst case in efficiency in the kernel. Reading from global memory is still the bottleneck of the application. We noted a severe drop in efficiency in saving the results back to the global memory by using sparse format. We found two principal reasons for this low efficiency: unbalanced access to global memory on the SpMV, and a higher percentage of stalls in the pipeline.

Regarding the stalls, in this architecture, the compiler could not hide global memory latency due to the increase of three reads from the global memory. Although ia and ja are constant vectors, fetching from global memory must be avoided.

From [Figure 35](#), we noted that changing matrix representation did not cause any improvement in BRAMS execution. We decided to check kernel execution and memory transfers.

Again, we compare the results with the dense version. From this, we could find what was causing the bad performance of Jacobi. Despite the lower execution time, we noted that there were more memory transfers between computations than in Architecture 1 due to CSR format.

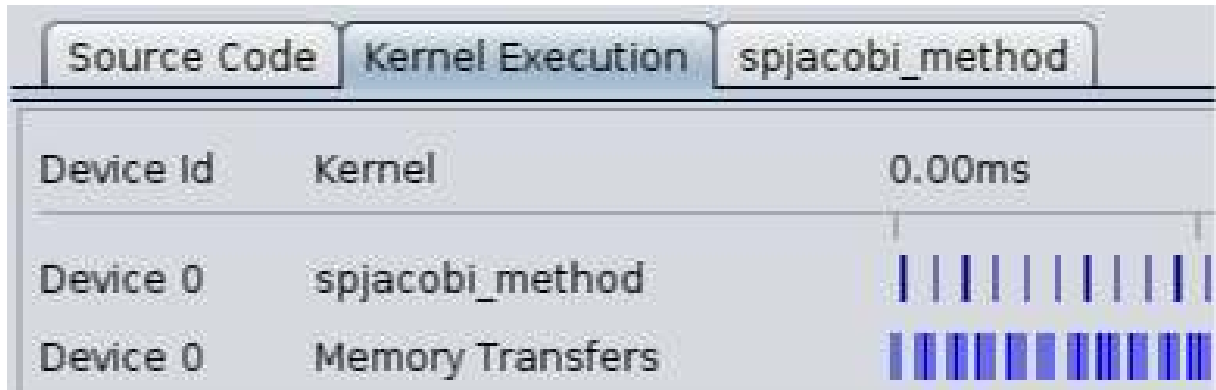


Figure 35 – Execution and memory transfer time of Jacobi multi-threaded sparse with Intel FPGA SDK profiling for kernels. Note that transfer time did not improve due to variable sparsity of the matrices.

In the dense representation, we needed to transfer four data structures, and now we send six data structures.

What draw our attention was the worst-case stall. In this version we improved in 6%. As expected, memory bandwidth is lower in the sparse architecture, since we need fewer data from global memory. Figure 36 shows these statistics.

Statistic	Measured	Optimal
Worse Case Stall (__global) %	3.29%	0%
Kernel Clock Frequency	307.8 MHz	na
Global BW (DDR:bank1)	1361.2 MB/s	12800 MB/s
Average Write Burst	1	16
Average Read Burst	3	16
Global BW (DDR:bank2 read-only)	1518.8 MB/s	12800 MB/s
Average Read Burst	3	16

Figure 36 – Statistics of Jacobi multi-threaded sparse with Intel FPGA SDK profiling for kernels.

As we did in the previous architecture, we evaluated the precision from this architecture. We used the same set of configuration for comparison, i.e. we used extrapolation to measure the error from the first block of the Rosenbrock Method obtained in Jacobi (hardware) compared to the first block obtained in the Sparse1.3a (software). As expected, our average error is $1.241371e - 19$; the same result from the previous architecture.

We also studied the impact of our new design on power consumption, as we mentioned earlier we cannot execute a weather forecasting for 24 hours due to time limitation. According to PowerPlay Power Analyzer Tool, this new architecture requires 14W. We show the summarized results in Table 8 and Table 9.

Table 8 – Results from Arch 2.

Area	Frequency	Time	Energy	Error
29%	260 MHz	~50 days	14 W	1.241e-19

Table 9 – Timing results from Arch 2.

CPU-FPGA	Execution	FPGA-CPU	Total Time
17597us	7846us	7863us	33306us

4.5 Results from Jacobi Single-threaded Sparse

This architecture has a significant improvement over the previous ones. Now we can compute the entire algorithm in hardware; this kernel is similar to a C to hardware.

We reused most of the knowledge of Architectures 1 and 2. At this point memory hierarchy was a solid concept, and we used the best practices for a single thread kernel. With this new concept of programming, new problems arose; we discuss the problems, solutions, and results from this architecture.

Our first version was a copy-paste of the Jacobi algorithm in software. We knew that there was room for plenty optimization, but we needed to perform the compilation to see the problems. Intel FPGA SDK allows us to compile only the intermediate object file; this is much faster than generating the hardware.

Compiling the kernel source into an intermediate representation is enough to provide a detailed report of loop pipelining. As we mentioned earlier, a single thread kernel pipelines the iterations of the loops. We show this report in [Figure 37](#) and discuss the improvements we made.

According to [Figure 37](#), copy from the current to the previous solution has II of 1; this the most efficient pipeline, it means that every cycle computes an iteration. Traversing each line of the matrix leads to a great result as well. We show an optimum pipeline in [Figure 38](#).

This ideal pipeline is not present in the loop responsible for traversing each NNZ element from the i^{th} row. The problem with this loop is the data dependency on the sum accumulator; as we cannot guarantee the same NNZ for each row, the pipeline must wait for the current iteration to finish before it starts the next one. This restriction also prevents us from using shift registers on the matrix-vector operation.

We show in [Figure 39](#), how it is the pipeline for matrix-vector accumulator; we use two iterations for simplicity. Note that this implies in an empty pipeline for some cycles, and it is responsible for reducing the performance of Jacobi.

We have a similar problem in the convergence calculation, but now there is a well-formed loop. We declared an array (conv) to avoid data dependency, and we used a shift register to guarantee an optimum II. For using a shift register in OpenCL, we fully unrolled the loop responsible for accumulating all the elements of conv. In [Figure 40](#), we show the report for the

Loops analysis				<input checked="" type="checkbox"/> Show fully unrolled loops
	Pipelined	II	Bottleneck	Details
Block1 (spjacobi_single_sem_otlmlzacao.cl:9)	No	n/a	n/a	Out-of-order Inner loop
Block2 (spjacobi_single_sem_otlmlzacao.cl:10)	Yes	1	n/a	
Block3 (spjacobi_single_sem_otlmlzacao.cl:13)	Yes	1	n/a	
Block4 (spjacobi_single_sem_otlmlzacao.cl:18)	Yes	11	11	Data dependency
Block6 (spjacobi_single_sem_otlmlzacao.cl:27)	Yes	11	11	Data dependency

spjacobi_single_sem_otlmlzacao.cl

```

1 #pragma OPENCL EXTENSION cl_khr_fp64 : enable
2 kernel
3 attribute__((reqd_work_group_size(1,1,1)))
4 void spjacobi_method1(__global const double *restrict data, __global
5 const double *restrict dataB, __global const int * restrict ia,
6 __global const int * restrict ja, __local
7 double *restrict outXold, __global
8 double *restrict outX){
9
10 int iteration = 0, maxIteration = 300;
11 while (iteration < maxIteration){
12 for (int i = 0; i < 47; i++){
13 outXold[i] = outX[i];
14 }
15 for (int i = 0; i < 47; i++){
16 int start = ia[i];
17 int end = ia[i + 1];
18 double sum = dataB[i];
19 double aii;
20 for (int j = start; j < end; j++){
21 sum += - dataA[j] * outXold[ja[j]];
22 if (ja[j] == 1){
23 aii = dataA[j];
24 }
25 outX[i] = sum/aii + outXold[i];
26 }
27 double conv;
28 for (int i = 0; i < 47; i++){
29 conv += pow((outX[i] - outXold[i]), 2);
30 }
31 if (sqrt(conv) < 1e-8) {
32 iteration = maxIteration;
33 }
34 iteration++;
35 }
36 }

```

Figure 37 – Pipeline report for Jacobi single-threaded sparse.

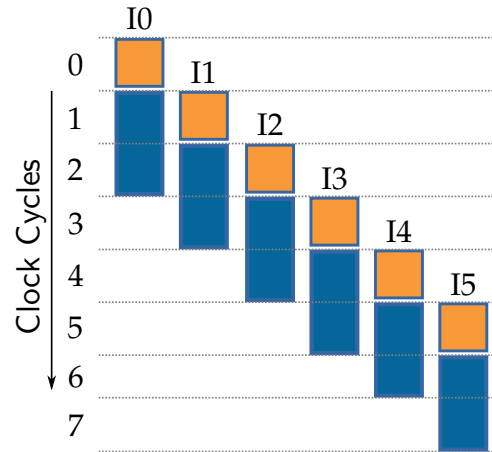


Figure 38 – Optimum pipeline with II of 1, figure from Intel (2016a).



Figure 39 – Matrix-vector pipeline with II of 11, based on Intel (2016a).

changes we made to this kernel.

These modification on the loop pipelines allowed us to reduce from 24 cycles to 14 cycles; an improvement of 42% compared to the initial solution. During this optimization stage, we noted that our kernel demanded one global write for each iteration of Jacobi; we improved that by using a local memory and performing a single write in the end of computation. With this solution, we want to measure stall, occupancy, efficiency, execution and transfer time of the kernel. We profiled the kernel to obtain accurate measures.

In Figure 41 we show the kernel efficiency. Despite the lower occupancy (18%) compared to the Architectures 1 and 2, we present optimum efficiency and stall for every pipeline. In this architecture, there is no bottleneck, which means that we achieved the best pipeline configuration. The worst-case stall is still very close to zero.

Using a shift register for convergence calculation led to an effective bandwidth. The

Loops analysis	Pipelined	II	Bottleneck	Details
Block1 (spjacobi_single_no_global_write.cl:9)	Yes	1	n/a	
Block2 (spjacobi_single_no_global_write.cl:13)	No	n/a	n/a	Out-of-order inner loop
Block3 (spjacobi_single_no_global_write.cl:14)	Yes	1	n/a	
Block4 (spjacobi_single_no_global_write.cl:17)	Yes	1	n/a	
Block5 (spjacobi_single_no_global_write.cl:22)	Yes	11	II	Data dependency
Block7 (spjacobi_single_no_global_write.cl:31)	Yes	1	n/a	
Fully unrolled loop (spjacobi_single_no_global_write.cl:35)	n/a	n/a	n/a	Unrolled by #pragma unroll
Block9 (spjacobi_single_no_global_write.cl:43)	Yes	1	n/a	

Show fully unrolled loops

spjacobi_single_no_global_write.cl

```

7   int iteration = 0, maxIteration = 300;
8   double outAux[47];
9   for (int i = 0; i < 47; i++){
10      outAux[i] = outX[i];
11  }
12
13  while (iteration < maxIteration){
14      for (int i = 0; i < 47; i++){
15          outX0d[i] = outAux[i];
16      }
17      for (int i = 0; i < 47; i++){
18          int start = ia[i];
19          int end = ia[i + 1];
20          double sum = dataB[i];
21          double aii;
22          for (int j = start; j < end; j++){
23              sum += - dataA[j] * outX0d[ja[j]];
24              if (ja[j] == i){
25                  aii = dataA[j];
26              }
27          }
28          outAux[i] = sum/aii + outX0d[i];
29      }
30      double conv[47], conv2;
31      for (int i = 0; i < 47; i++){
32          conv[i] = pow((outAux[i] - outX0d[i]), 2);
33      }
34      #pragma unroll
35      for (int i = 0; i < 47; i++){
36          conv2 += conv[i];
37      }
38      if (sqrt(conv2) < 1e-8) {
39          iteration = maxIteration;
40      }
41      iteration++;
42  }
43  for (int i = 0; i < 47; i++){
44      outX[i] = outAux[i];
45  }
46  }
47  }
  
```

Figure 40 – Pipeline report for Jacobi single-threaded sparse optimized.

Line #	Source: spjacobi_single_no_global_write.cl	Attributes	Stall%	Occupancy%	Bandwidth
8	double outAux[47];				
9	for (int i = 0; i < 47; i++){				
10	outAux[i] = outX[i];	0: __global{DDR},read	0: 0.01%	0: 0.1%	0: 0.6MB/s, 100.00%Efficiency
11	}				
12					
13	while (iteration < maxiteration){				
14	for (int i = 0; i < 47; i++){				
15	outXOld[i] = outAux[i];	0: __local,read	0: 0.0%	0: 3.6%	0: --
16	}				
17	for (int i = 0; i < 47; i++){				
18	int start = ia[i];	(__global{DDR},read)	(0.01%)	(0.1%)	(0.1MB/s, 100.00%Efficiency)
19	int end = ia[i + 1];	(__global{DDR},read)	(0.02%)	(3.6%)	(0.4MB/s, 100.00%Efficiency)
20	double sum = dataB[i];	(__global{DDR},read)	(0.04%)	(3.6%)	(0.6MB/s, 100.00%Efficiency)
21	double aii;				
22	for (int j = start; j < end; j++){				
23	sum += - dataA[j] * outXOld[ja[j]];	1: __global{DDR},read	1: 0.12%	1: 18.3%	1: 15.8MB/s, 100.00%Efficiency
24	if (ja[j] == i){				
25	aii = dataA[j];				
26	}				
27	}				
28	outAux[i] = sum/aii + outXOld[i];	0: __local,read	0: 0.0%	0: 3.6%	0: --
29	}				
30	double conv[47], conv2;				
31	for (int i = 0; i < 47; i++){				
32	conv[i] = pow((outAux[i] - outXOld[i]), 2);	0: __local,read	0: 0.0%	0: 3.6%	0: --
33	}				
34	#pragma unroll				
35	for (int i = 0; i < 47; i++){				
36	conv2 += conv[i];	(__local,read)	(0.0%)	(0.1%)	(--)
37	}				
38	if (sqrt(conv2) < 1e-8) {				
39	iteration = maxiteration;				
40	}				
41	iteration++;				
42	}				
43	for (int i = 0; i < 47; i++){				
44	outX[i] = outAux[i];	0: __local,read	0: 0.0%	0: 0.1%	0: --

Figure 41 – Efficiency for Jacobi single-threaded sparse. After the modifications, all the pipelines shows 100% of efficiency and almost zero stall.

same applies to the stalls in the pipeline; in the profiling, we can see numbers very close to zero. Besides the great results for the pipelines and stalls, we cannot infer the same about occupancy.

The changes in the architecture showed that occupancy is still a problem. That is not a design related problem, but a data problem. We do not provide enough data for the kernel, resulting in an idle time in the kernel; in future work, we intend to change BRAMS structure related to the chemical reactivity — spack data structure. In this manner, we will be able to improve occupancy.

Even with low occupancy, we had an improvement with this design compared to the previous ones. According to our results for a 24-hour weather forecasting, we improved the computational time in 63 times. We show in [Figure 42](#) execution and memory transfers time.

Here we present an appropriate balance between execution and memory transfers. This result relies on the improved Jacobi method; we implemented it 100% in hardware, in this manner Jacobi only communicates with the host to receive and send data. According to our measures, processing an entire linear system takes in average *1ms*.

As we mentioned before, our design suffers from the lack of massive data to process. [Figure 43](#) supports this affirmation by showing how much data is necessary during execution (measured column); in the same figure, we can see that the burst write is six times better in this

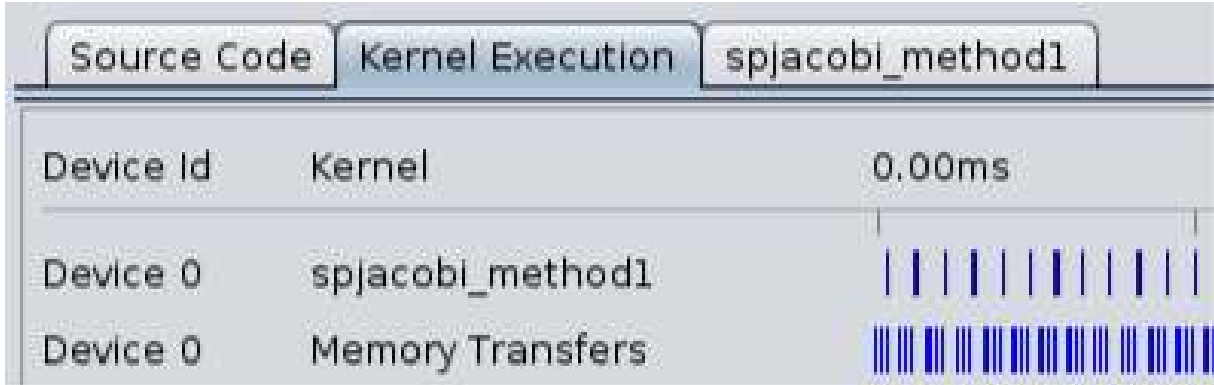


Figure 42 – Execution and memory transfers time for Jacobi single-threaded sparse. Each bar in `spjacobi_method1` means a complete execution over a matrix.

architecture than in Architecture 2, and two times better than Architecture 1.

Statistic	Measured	Optimal
Worse Case Stall (<code>__global</code>) %	0.12%	0%
Kernel Clock Frequency	284.3 MHz	na
Global BW (DDR:bank1)	13.6 MB/s	12800 MB/s
Average Write Burst	6	16
Average Read Burst	1	16
Global BW (DDR:bank2 read-only)	23.7 MB/s	12800 MB/s
Average Read Burst	1	16

Figure 43 – Statistics for Jacobi single-threaded sparse.

Once we improved time execution, we could perform comparisons with software regarding the precision of the floating point. For such comparison, we retrieved the first block of error of the Rosenbrock Method in software and in hardware every timestep, then we measured the difference between the results. We measured the error for 1440 timesteps (24-hour weather forecasting), and we obtained an average error of $-6.277075e - 09$. If we compare in the same conditions as the previous architectures, we obtain an average error of $8.027268e - 20$, i.e. $10 \times$ smaller than Architectures 1 and 2.

Jacobi impacts Rosenbrock convergence rate since it depends on the error of fourth stage. In the Literature, we find several works supporting that Jacobi method has a slow convergence rate (TAMULI *et al.*, 2015; KHUSHPREETKAUR, 2012). We decided to take advantage of this slow rate convergence to improve throughput by using double buffering scheme. In Figure 42, there are some matrices whose execution time is higher than memory transfer.

According to our experiments, we could not use double buffering on Intel FPGA SDK correctly. By using this scheme, we noted that data from different queues were overlapping the

pipeline, which returned a wrong result. No warning or error during compilation occurred, we found these errors during the measuring of Rosenbrock error.

We decided to make a copy of the kernel and then distribute them over the queues, in our case, we use two queues. Our first attempt was not successful due to space limitation of the FPGA; we had to downgrade our design to make the replication fit the FPGA.

During the execution of BRAMS with the downgraded version, we noted the same problems we had before. According to our experiments, sharing the FPGA is not possible when using OpenCL for Intel devices (a similar problem happened when we tried to share the FPGA for more than one process, see [Section 4.2](#)).

After all the possible optimizations, we decided to measure the energy required for processing the chemical reactivity in FPGA. We could make a more accurate test than the previous architectures since we could perform a complete execution of BRAMS for a 24-hour weather forecasting.

According to our results, this architecture requires 15W. So far, this architecture is the most energy efficient and fastest version we implemented by using 15% more of resources of the FPGA. We summarize the results in [Table 10](#) and in [Table 11](#).

Table 10 – Results from Arch 3.

Area	Frequency	Time	Energy	Error
34%	269 MHz	~19 hours	15 W	8.027e-20

Table 11 – Tmining results from Arch 3.

CPU-FPGA	Execution	FPGA-CPU	Total Time
92us	912us	9us	1013us

4.6 Results from Jacobi Single-threaded Dense

In the Arch 3, we noticed that NNZ is a problem in CSR sparse format since we cannot infer an optimum pipeline. To avoid this problem, we implemented a dense version of Arch 3; we called it Arch 4. We do not present any report for this architecture since we it did not improve the time execution. The compiler unrolls automatically the internal loop to process more data in parallel, this feature can be disabled, this is why the hardware takes 45% of the available hardware. Unrolling the loop also decreases hardware frequency, it needs 239 Mhz.

In this manner, we can conclude that sparse format is still the most suitable format for our problem. We show the results in [Table 12](#), as we can see in this table, transferring from CPU to FPGA is faster using dense format. This behavior shows that transferring small chunks of data is slower than sending bigger ones due to the delay of the OpenCL API.

Table 12 – Timing results from Arch 4.

CPU-FPGA	Execution	FPGA-CPU	Total Time
78us	3090us	16us	3184us

4.7 Results from Sparse1.3a

We also executed BRAMS with a single process with Sparse1.3a for a 24-hour weather forecasting. We compare execution time from the software library with our Arch 3.

According to the results, the software versions takes about one hour and fifteen minutes to complete the execution. The same execution in hardware takes about 19 hours to execute, i.e. our hardware demands $15\times$ more time execution, which is prohibitive in practice.

This poor execution time is related to several factors: (a) matrix size; (b) communication; and (c) one process per FPGA. Batched execution of linear algebra operations on small matrices are still a problem, and high performance accelerator remains a challenging problem (DONG *et al.*, 2016).

The focus of this project was to migrate a snippet of BRAMS to a heterogeneous machine with FPGA, which was successfully performed. Time execution is not critical at this moment. In Table 13, we show the main characteristics of each implementation, and their advantages and disadvantages. In Table 15, we compare the transfer and execution time for each architecture.

Table 13 – Table of comparison among implementations.

Features	A	B	C	D	E	Advantages	Disadvantage
Arch 1	X		X	X		Easy to understand and program.	Too much communication between CPU-FPGA.
Arch 2	X		X	X		Pipeline never computes zeros.	It is hard to define a suitable NDRange.
Arch 3	X		X	X		100% of Jacobi in hardware.	Performance loss due to data representation.
Arch 4	X		X	X		No load imbalance.	Performance loss compared to Arch 3.
Sw		X	X		X	Fast and suitable for CPUs.	It is not suitable for parallelism.

Features:

- A: Parallel algorithm;
- B: MPI Support;
- C: Coupled to BRAMS;
- D: Iterative method;
- E: Direct Method;

Table 15 – Comparison among architectures

Arch	CPU-FPGA	Execution	FPGA-CPU	Total Time
1	11686us	9153us	7806us	28645us
2	17597us	7846us	7863us	33306us
3	92us	912us	9us	1013us
4	78us	3090us	16us	3184us

CONCLUSION

In this project, we explore the possibility to use heterogeneous computing to solve the chem term. As we showed in the related works, coupling the chemical reactivity to BRAMS is a complex task that we could perform by using OpenCL. We used this framework to implement the dense and sparse version of Jacobi, an iterative linear solver.

We provide a parallel Jacobi suitable for FPGAs pipelining. By using FPGA, we could implement 100% of the algorithm on the device side and still maintain it coupled to BRAMS, to the best of our knowledge, this is the first work that couples FPGA to BRAMS. According to our tests, we proved the program presents the same behavior and accuracy of the software.

Such algorithm is critical to the solution of the chem term of the mass continuity equation; the chemical term became the most expensive when the developers included the gas chemical module. We obtained the same results through profilings techniques applied to BRAMS, such as Gprof. Currently, BRAMS executes only in CPU with MPI parallelism.

With this project, we could prove that porting parts of BRAMS to heterogeneous computing — CPU-FPGA — is possible. Our work allows the scientific community to explore more parallel solution using FPGAs.

5.1 Limitations

As we presented in [Section 4.2](#), our design does not allow more than one process to access the same device. The sparse format we chose imposes a severe drop of performance in pipeline execution. Performance is also a problem since our best architecture is $15\times$ slower than software.

CSR format imposes a load imbalance on the computation, and such problem imposes a higher latency in the pipeline. With this format, it is not possible to improve II of SpMV operation since the sparsity of the matrices varies from 8% up to 25%.

5.2 Future Work

In future work, we intend to solve the limitations in our design. Our first target is the Rosenbrock method, which we could perform the stages of the method in hardware, as we saw in [Figure 2](#), all Rosenbrock stages require minor modifications and perform the same computation. Such implementation could be carried out if there is enough space in FPGA area.

We would also need to study the I/O involved in this process if this is a suitable solution. Regarding the sparse format, we could explore the viability of other sparse formats that could perform in parallel. One of the options is the Ellpack format, which does not impose a load imbalance. Although it seems a perfect solution, such format could lead to huge amount of additional memory and computation.

During BRAMS execution, we noted that much of the communication could be avoided. Instead of sending a single matrix to process and receiving a single result, we could process the entire block by sending all the data related to the block and manage them in hardware.

Another possible solution is to explore the serialization of the MPI processes, i.e. when they reach FPGA execution only one of the process could perform the computation at a time. For this, we could use a master-slave approach similar to what already happens in BRAMS, where just the master has access to the device.

A harder optimization requires the modification of BRAMS source code related to chemical reactivity data structure — spack data structure. Fortran90 does not align components of derived type, i.e. components from the data structure. A deeper study of this structure could boost BRAMS performance.

BIBLIOGRAPHY

ACCELEWARE. **OpenCL on FPGAs for GPU Programmers**. [S.l.], 2014. Available at: <https://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf>. Accessed on: May 12, 2017. Cited on page 52.

ACKERMAN, P.; ACKERMAN, S.; KNOX, J. **Meteorology**. Jones & Bartlett Learning, LLC, 2013. ISBN 9781284027389. Available at: <https://books.google.com.br/books?id=qWcrAQAQAQBAJ>. Cited on page 30.

AHMAD, S.; BOPANA, V.; GANUSOV, I.; KATHAIL, V.; RAJAGOPALAN, V.; WITTIG, R. A 16-nm multiprocessing system-on-chip field-programmable gate array platform. **IEEE Micro**, IEEE, v. 36, n. 2, p. 48–62, 2016. Cited 2 times on pages 13 and 45.

ALISSON, E. **Sistema faz previsões simultâneas de tempo e qualidade do ar na América do Sul**. [S.l.], 2016. Available at: http://agencia.fapesp.br/sistema_faz_previsoes_simultaneas_de_tempo_e_qualidade_do_ar_na_america_do_sul/22921/. Accessed on: Mar 20, 2017. Cited on page 55.

ALTERA. **Implementing FPGA Design with the OpenCL Standard**. [S.l.], 2013. Available at: https://www.altera.com/en_US/pdfs/literature/wp/wp-01173-opencl.pdf. Accessed on: May 4, 2017. Cited 2 times on pages 13 and 42.

AN, J.; WANG, D. Efficient one-sided jacobi svd computation on amd gpu using opencl. In: **2016 IEEE 13th International Conference on Signal Processing (ICSP)**. [S.l.: s.n.], 2016. p. 491–495. ISSN 2164-5221. Cited on page 51.

ANTON, H.; RORRES, C. **Elementary Linear Algebra: Applications Version, 11th Edition**. Wiley Global Education, 2013. ISBN 9781118879160. Available at: <https://books.google.com.br/books?id=loRbAgAAQBAJ>. Cited on page 35.

BELL, N.; GARLAND, M. **Efficient sparse matrix-vector multiplication on CUDA**. [S.l.], 2008. Cited on page 78.

BINDEL, D.; GOODMAN, J. Principles of scientific computing linear algebra ii, algorithms. 2006. Cited on page 36.

BITTWARE, I. **S5-PCIe-HQ**. [S.l.], 2015. 57 p. Cited 2 times on pages 13 and 46.

BLICKLE, T.; TEICH, J.; THIELE, L. System-level synthesis using evolutionary algorithms. **Design Automation for Embedded Systems**, Springer, v. 3, n. 1, p. 23–58, 1998. Cited on page 48.

BOBDA, C. **Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications**. Springer Netherlands, 2007. ISBN 9781402061004. Available at: https://books.google.com.br/books?id=_cNSgjR32LkC. Cited on page 44.

BORKAR, S.; CHIEN, A. A. The future of microprocessors. **Communications of the ACM**, ACM, v. 54, n. 5, p. 67–77, 2011. Cited on page 28.

BOUT, D. V. **FPGAs?! Now What?** [s.n.], 2011. Available at: <http://www.xess.com/static/media/appnotes/FpgasNowWhatBook.pdf>. Cited on page 45.

BRAVO, I.; JIMENEZ, P.; MAZO, M.; LAZARO, J. L.; GARDEL, A. Implementation in fpgas of jacobi method to solve the eigenvalue and eigenvector problem. In: **2006 International Conference on Field Programmable Logic and Applications**. [S.l.: s.n.], 2006. p. 1–4. ISSN 1946-147X. Cited on page 49.

BUCHTY, R.; HEUVELINE, V.; KARL, W.; WEISS, J.-P. A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 24, n. 7, p. 663–675, 2012. Cited on page 38.

CHE, S.; LI, J.; SHEAFFER, J. W.; SKADRON, K.; LACH, J. Accelerating compute-intensive applications with gpus and fpgas. In: IEEE. **Application Specific Processors, 2008. SASP 2008. Symposium on**. [S.l.], 2008. p. 101–107. Cited on page 28.

CHENG, J.; GROSSMAN, M.; MCKERCHER, T. **Professional CUDA C Programming**. Wiley, 2014. (Wrox : Programmer to Programmer). ISBN 9781118739310. Available at: https://books.google.com.br/books?id=Jgx_BAAAQBAJ. Cited on page 67.

CHU, P. **FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version**. Wiley, 2011. ISBN 9781118210604. Available at: <https://books.google.com.br/books?id=nXdbDRUUCyUC>. Cited on page 44.

CONG, J.; ZOU, Y. Fpga-based hardware acceleration of lithographic aerial image simulation. **ACM Transactions on Reconfigurable Technology and Systems (TRETS)**, ACM, v. 2, n. 3, p. 17, 2009. Cited on page 28.

CRASSIER, V.; SUHRE, K.; TULET, P.; ROSSET, R. Development of a reduced chemical scheme for use in mesoscale meteorological models. **Atmospheric Environment**, Elsevier, v. 34, n. 16, p. 2633–2644, 2000. Cited on page 31.

CZAJKOWSKI, T. S.; AYDONAT, U.; DENISENKO, D.; FREEMAN, J.; KINSNER, M.; NETO, D.; WONG, J.; YIANNACOURAS, P.; SINGH, D. P. From opencl to high-performance hardware on fpgas. In: IEEE. **Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on**. [S.l.], 2012. p. 531–534. Cited on page 41.

CZAJKOWSKI, T. S.; NETO, D.; KINSNER, M.; AYDONAT, U.; WONG, J.; DENISENKO, D.; YIANNACOURAS, P.; FREEMAN, J.; SINGH, D. P.; BROWN, S. D. Opencl for fpgas: Prototyping a compiler. In: **Int'l Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)**. [S.l.: s.n.], 2012. p. 3–12. Cited 3 times on pages 13, 43, and 44.

DAGA, V.; GOVINDU, G.; PRASANNA, V.; GANGADHARAPALLI, S.; SRIDHAR, V. Efficient floating-point based block lu decomposition on fpgas. In: **International Conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas**. [S.l.: s.n.], 2004. p. 21–24. Cited on page 49.

DAVISON, M. **Shallow/Deep Convection**. [S.l.], 1999. Available at: <http://origin.wpc.ncep.noaa.gov/international/training/deep/sld001.htm>. Accessed on: May 8, 2017. Cited on page 30.

DINIGROUP. **FPGA Selection Guide**. [S.l.], 2017. Available at: <http://www.dinigroup.com/product/common/DINI_selection_guide_v540.pdf>. Accessed on: May 4, 2017. Cited on page 46.

DONG, T.; HAIDAR, A.; LUSZCZEK, P.; TOMOV, S.; ABDELFAH, A.; DONGARRA, J. Magma batched: A batched blas approach for small matrix factorizations and applications on gpus. Aug 2016. Cited on page 87.

ELLER, P.; SINGH, K.; SANDU, A. Development and acceleration of parallel chemical transport models. In: SOCIETY FOR COMPUTER SIMULATION INTERNATIONAL. **Proceedings of the 2010 Spring Simulation Multiconference**. [S.l.], 2010. p. 90. Cited 2 times on pages 49 and 52.

ERNST, R.; HENKEL, J.; BENNER, T. Hardware-software cosynthesis for microcontrollers. **IEEE Design Test of Computers**, v. 10, n. 4, p. 64–75, Dec 1993. ISSN 0740-7475. Cited on page 48.

FAZENDA, A. L.; ENARI, E. H.; RODRIGUES, L. F.; PANETTA, J. Towards production code effective portability among vector machines and microprocessor-based architectures. In: **IEEE. Computer Architecture and High Performance Computing, 2006. SBAC-PAD'06. 18TH International Symposium on**. [S.l.], 2006. p. 11–20. Cited on page 51.

FAZENDA, A. L.; RODRIGUES, E. R.; TOMITA, S. S.; PANETTA, J.; MENDES, C. L. Improving the scalability of an operational scientific application in a large multi-core cluster. In: **2012 13th Symposium on Computer Systems**. [S.l.: s.n.], 2012. p. 126–132. Cited on page 34.

FERNANDES, A. d. A. **Paralelização do Termo de Reatividade Química do Modelo Ambiental CCATT-BRAMS utilizando um Solver Baseado em Estimação Linear Ótima**. 76 p. Dissertação (Mestrado) — Instituto Nacional de Pesquisas Espaciais, 2014. Master's thesis at INPE-SP. Cited 4 times on pages 31, 51, 55, and 62.

FOERTSCH, J.; JOHNSON, J.; NAGVAJARA, P. Jacobi load flow accelerator using fpga. In: **Proceedings of the 37th Annual North American Power Symposium, 2005**. [S.l.: s.n.], 2005. p. 448–454. Cited on page 49.

FREITAS, S.; LONGO, K.; DIAS, M. S.; CHATFIELD, R.; DIAS, P. S.; ARTAXO, P.; ANDRAE, M.; GRELL, G.; RODRIGUES, L.; FAZENDA, A. *et al.* The coupled aerosol and tracer transport model to the brazilian developments on the regional atmospheric modeling system (catt-brams)—part 1: Model description and evaluation. **Atmospheric Chemistry and Physics**, Copernicus GmbH, v. 9, n. 8, p. 2843–2861, 2009. Cited on page 29.

FREITAS, S.; LONGO, K.; TRENTMANN, J.; LATHAM, D. Technical note: Sensitivity of 1-d smoke plume rise models to the inclusion of environmental wind drag. **Atmospheric Chemistry and Physics**, Copernicus GmbH, v. 10, n. 2, p. 585–594, 2010. Cited on page 30.

FU, H.; GAN, L.; YANG, C.; XUE, W.; WANG, L.; WANG, X.; HUANG, X.; YANG, G. Solving global shallow water equations on heterogeneous supercomputers. **PloS one**, Public Library of Science, v. 12, n. 3, p. e0172583, 2017. Cited on page 52.

GAILLY, J.-I.; ADLER, M. **A Massively Spiffy Yet Delicately Unobtrusive Compression Library (Also Free, Not to Mention Unencumbered by Patents)**. [S.l.], 2015. Available at: <<http://www.zlib.net/>>. Accessed on: Dec. 15, 2015. Cited on page 34.

GALLERY, R. Hardware/software codesign. **The ITB Journal**, v. 4, n. 1, p. 5, 2015. Cited 2 times on pages 46 and 47.

GOLUB, G.; LOAN, C. V. **Matrix Computations**. Johns Hopkins University Press, 2013. (Johns Hopkins Studies in the Mathematical Sciences). ISBN 9781421407944. Available at: <<https://books.google.com.br/books?id=X5YfsuCWpxMC>>. Cited on page 36.

GOMES, G. A. A. Linear solvers for stable fluids: Gpu vs cpu. **17th EncontroPortugues de Computacao Grafica (EPCG09)**, p. 145–153, 2009. Cited on page 51.

GRAHAM, S.; PARKINSON, C.; CHAHINE, M. **Weather Forecasting Through the Ages**. [S.l.], 2002. Available at: <<http://earthobservatory.nasa.gov/Features/WxForecasting/wx.php>>. Accessed on: July 30, 2015. Cited on page 25.

GRØNØS, S. Vilhelm bjerknæs' vision for scientific weather prediction. **The Nordic Seas: An Integrated Perspective**, Wiley Online Library, p. 357–366, 2005. Cited on page 26.

GROPP, W.; HOEFLER, T.; LUSK, E.; THAKUR, R. **Using Advanced MPI: Modern Features of the Message-Passing Interface**. MIT Press, 2014. (Computer science & intelligent systems). ISBN 9780262527637. Available at: <<https://books.google.com.br/books?id=Po5IBQAAQBAJ>>. Cited on page 35.

GROUP, H. **HDF5 Technologies**. [S.l.], 2011. Available at: <https://www.hdfgroup.org/about/hdf_technologies.html>. Accessed on: Dec. 15, 2015. Cited on page 34.

GUPTA, P. **Xeon+FPGA Platform for the Data Center**. [S.l.], 2015. Available at: <<https://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf>>. Accessed on: Apr 10, 2017. Cited on page 46.

GUPTA, R. K.; MICHELI, G. D. Hardware-software cosynthesis for digital systems. **IEEE Design Test of Computers**, v. 10, n. 3, p. 29–41, Sept 1993. ISSN 0740-7475. Cited on page 48.

GÁCITA, M. S. **Estudos Numéricos de Química Atmosférica para a região do Caribe e América Central com Ênfase em Cuba**. Dissertação (Mestrado) — Instituto Nacional de Pesquisas Espaciais - INPE, São José dos Campos - SP - Brasil, 2011. Cited on page 31.

HARRIS, M. **How to Optimize Data Transfers in CUDA C/C++**. [S.l.], 2012. Available at: <<https://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/>>. Accessed on: Apr 3, 2017. Cited 2 times on pages 14 and 67.

HENKEL, M. **21st Century Homestead: Sustainable Agriculture II: Farming and Natural Resources**. Lulu.com, 2015. ISBN 9781312939684. Available at: <<https://books.google.com.br/books?id=jmHxCQAAQBAJ>>. Cited on page 25.

HRZ. **MPI + OpenCL Altera**. [S.l.], 2017. Available at: <<http://www.alteraforum.com/forum/showthread.php?t=55035>>. Accessed on: Mar 6, 2017. Cited on page 71.

INPE/CPTEC. **Model Description**. [S.l.], 2015. Available at: <<http://brams.cptec.inpe.br/>>. Accessed on: Dec. 16, 2015. Cited on page 29.

INTEL. **Intel FPGA SDK for OpenCL – Best Practices Guide**. [S.l.], 2016. Available at: <https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf>. Accessed on: Mar 29, 2017. Cited 8 times on pages 13, 15, 42, 43, 57, 64, 74, and 82.

_____. **Intel FPGA SDK for OpenCL – Programming Guide**. [S.l.], 2016. Available at: https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf. Accessed on: May 4, 2017. Cited on page 42.

JANIK, I.; TANG, Q.; KHALID, M. An overview of altera sdk for opencl: A user perspective. In: IEEE. **Electrical and Computer Engineering (CCECE), 2015 IEEE 28th Canadian Conference on**. [S.l.], 2015. p. 559–564. Cited 2 times on pages 43 and 44.

KAPRE, N.; DEHON, A. Parallelizing sparse matrix solve for spice circuit simulation using fpgas. In: IEEE. **Field-Programmable Technology, 2009. FPT 2009. International Conference on**. [S.l.], 2009. p. 190–198. Cited on page 48.

_____. *rmSPICE²*: Spatial processors interconnected for concurrent execution for accelerating the spice circuit simulator using an fpga. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 31, n. 1, p. 9–22, Jan 2012. ISSN 0278-0070. Cited on page 49.

KARNIADAKIS, G.; KIRBY, R. **Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and their Implementation**. Cambridge University Press, 2003. ISBN 9781107494770. Available at: <https://books.google.com.br/books?id=Py8XAgAAQBAJ>. Cited on page 35.

KASBAH, S. J.; DAMAJ, I. W. The jacobi method in reconfigurable hardware. In: **World Congress on Engineering**. [S.l.: s.n.], 2007. p. 823–827. Cited on page 49.

KHUSHPREETKAUR, H. Convergence of jacobi and gauss-seidel method and error reduction factor. **IOSR Journal of Mathematics (IOSRJM)**, v. 2, p. 20–23, 2012. Cited on page 85.

KIRK, D. B.; WEN-MEI, W. H. **Programming massively parallel processors: a hands-on approach**. [S.l.]: Newnes, 2012. Cited on page 27.

KRISHNAIYER, R. **Data Alignment to Assist Vectorization**. [S.l.], 2015. Available at: <https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization>. Accessed on: Mar 21, 2017. Cited on page 62.

KUNDERT, K. S.; SANGIOVANNI-VINCENTELLI, A. **Sparse1.3**. [S.l.], 1988. Available at: <http://web.cs.ucla.edu/classes/CS258G/sis-1.3/sis/linsolv/>. Accessed on: Oct. 28, 2015. Cited 2 times on pages 32 and 36.

LABORATORY, E. S. R. **Regional Modeling**. [S.l.], 2015. Available at: <http://www.esrl.noaa.gov/research/themes/regional>. Accessed on: July 30, 2015. Cited 2 times on pages 25 and 26.

LAMBERS, J. **Jacobi Methods**. [S.l.], 2010. Available at: <http://web.stanford.edu/class/cme335/lecture7.pdf>. Accessed on: May 11, 2017. Cited on page 51.

LARSON, R. **Elementary Linear Algebra**. Cengage Learning, 2016. ISBN 9781305887824. Available at: <https://books.google.com.br/books?id=2sQaCgAAQBAJ>. Cited on page 35.

LINFORD, J. C.; MICHALAKES, J.; VACHHARAJANI, M.; SANDU, A. Multi-core acceleration of chemical kinetics for simulation and prediction. In: IEEE. **High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on**. [S.l.], 2009. p. 1–11. Cited on page 55.

LINFORD, J. C.; SANDU, A. Vector stream processing for effective application of heterogeneous parallelism. In: **Proceedings of the 2009 ACM Symposium on Applied Computing**. New York, NY, USA: ACM, 2009. (SAC '09), p. 976–980. ISBN 978-1-60558-166-8. Available at: <http://doi.acm.org/10.1145/1529282.1529496>. Cited on page 51.

LONGO, K.; FREITAS, S.; PIRRE, M.; MARÉCAL, V.; RODRIGUES, L.; PANETTA, J.; ALONSO, M.; ROSÁRIO, N.; MOREIRA, D.; GÁCITA, M. *et al.* The chemistry catts-rams model (ccatts-rams 4.5): a regional atmospheric model system for integrated air quality and weather forecasting and research. **Model Dev. Discuss**, v. 6, p. 1173–1222, 2013. Cited 4 times on pages 13, 29, 30, and 31.

LYNCH, P. The origins of computer weather prediction and climate modeling. **Journal of Computational Physics**, Elsevier, v. 227, n. 7, p. 3431–3444, 2008. Cited on page 26.

MCEWEN, J.; EYERS, D. **gzip1.0b1 Data compression on the sphere**. [S.l.], 2011. Available at: <http://www.jasonmcewen.org/codes/gzip/>. Accessed on: Dec. 15, 2015. Cited on page 34.

MICHALAKES, J.; VACHHRAJANI, M. Gpu acceleration of numerical weather prediction. **Parallel Processing Letters**, v. 18, n. 04, p. 531–548, 2008. Available at: <http://www.worldscientific.com/doi/abs/10.1142/S0129626408003557>. Cited on page 51.

MOORE ANDREW; WILSON, R. **FPGAs for Dummies**. Wiley, 2017. ISBN 978-1-119-39047-3. Available at: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/misc/fpgas_for_dummies_ebook.pdf. Cited on page 44.

MOREIRA, D.; FREITAS, S.; BONATTI, J.; MERCADO, L.; ROSÁRIO, N.; LONGO, K.; MILLER, J.; GLOOR, M.; GATTI, L. Coupling between the jules land-surface scheme and the ccatts-rams atmospheric chemistry model (jules-ccatts-rams1. 0): applications to numerical weather forecasting and the co 2 budget in south america. **Geoscientific Model Development**, Copernicus GmbH, v. 6, n. 4, p. 1243–1259, 2013. Cited on page 32.

MORRIS, G. R.; PRASANNA, V. K. An fpga-based floating-point jacobi iterative solver. In: IEEE. **Parallel Architectures, Algorithms and Networks, 2005. ISPAN 2005. Proceedings. 8th International Symposium on**. [S.l.], 2005. p. 8–pp. Cited 3 times on pages 36, 49, and 57.

MPICH. **MPICH Overview**. [S.l.], 2015. Available at: <https://www.mpich.org/about/overview/>. Accessed on: Dec. 15, 2015. Cited on page 34.

MUNSHI, A. **The OpenCL Specification**. [S.l.], 2009. Available at: <https://www.khronos.org/registry/OpenCL/specs/ocl1.0.pdf>. Accessed on: Mar 21, 2017. Cited 4 times on pages 13, 40, 41, and 60.

MUNSHI, A.; GASTER, B.; MATTSON, T. G.; GINSBURG, D. **OpenCL programming guide**. [S.l.]: Pearson Education, 2011. Cited 4 times on pages 37, 38, 40, and 61.

NIELSEN, F. **Introduction to HPC with MPI for Data Science**. Springer International Publishing, 2016. (Undergraduate Topics in Computer Science). ISBN 9783319219035. Available at: <https://books.google.com.br/books?id=eDiFCwAAQBAJ>. Cited on page 35.

OSTHOFF, C.; SCHEPKE, C.; VILASBÔAS, F.; BOITO, F.; PANETTA, J.; PILLA, L.; MAILLARD, N.; GRUNMANN, P.; DIAS, P. L. S.; LOPES, P. P. *et al.* **Improving Atmospheric Model Performance on a Multi-Core Cluster System**. [S.l.]: INTECH Open Access Publisher, 2012. Cited on page 26.

PANETTA, J. **Production**. [S.l.], 2015. Available at: <<http://www.cesup.ufrgs.br/videos/jairo-panetta/view>>. Accessed on: Oct. 28, 2015. Cited on page 26.

PATTERSON, D.; HENNESSY, J. **Computer Organization and Design: The Hardware/software Interface**. Morgan Kaufmann, 2012. (Morgan Kaufmann Series in Computer Graphics). ISBN 9780123747501. Available at: <<https://books.google.com.br/books?id=DMxe9AI4-9gC>>. Cited 3 times on pages 13, 37, and 38.

PENG, R. **Algorithm design using spectral graph theory**. Tese (Doutorado) — Microsoft Research, 2013. Cited on page 36.

PRASANNA, V. K.; MORRIS, G. R. Sparse matrix computations on reconfigurable hardware. **Computer**, v. 40, n. 3, p. 58–64, March 2007. ISSN 0018-9162. Cited on page 49.

RANDALL, D. A. **An Introduction to Atmospheric Modeling**. [s.n.], 2013. Available at: <<http://kiwi.atmos.colostate.edu/group/dave/at604.html>>. Cited on page 30.

RAYMOND, E. S. **The Cathedral & the Bazaar: Musings on linux and open source by an accidental revolutionary**. [S.l.]: " O'Reilly Media, Inc.", 2001. Cited on page 29.

REW, R.; DAVIS, G. Netcdf: an interface for scientific data access. **IEEE Computer Graphics and Applications**, v. 10, n. 4, p. 76–82, July 1990. ISSN 0272-1716. Cited on page 32.

REW, R. K. **What Is netCDF?** [S.l.], 2015. Available at: <<http://www.unidata.ucar.edu/software/netcdf/docs/faq.html#whatisit>>. Accessed on: Dec. 15, 2015. Cited on page 32.

RODRIGUES, E. R.; MADRUGA, F. L.; NAVAUX, P. O.; PANETTA, J. Multi-core aware process mapping and its impact on communication overhead of parallel applications. In: **IEEE Computers and Communications, 2009. ISCC 2009. IEEE Symposium on**. [S.l.], 2009. p. 811–817. Cited on page 57.

RUAN, H.; HUANG, X.; FU, H.; YANG, G. Jacobi solver: A fast fpga-based engine system for jacobi method. **Research Journal of Applied Sciences, Engineering and Technology**, March 2013. ISSN 2040-7459. Cited on page 50.

SCHAUMONT, P. **A practical introduction to hardware/software codesign**. [S.l.]: Springer Science & Business Media, 2012. Cited 3 times on pages 13, 47, and 48.

SCHMID, M.; REICHE, O.; SCHMITT, C.; HANNIG, F.; TEICH, J. Code generation for high-level synthesis of multiresolution applications on fpgas. **arXiv preprint arXiv:1408.4721**, 2014. Cited 2 times on pages 50 and 51.

SILVA, E. Pereira da. **Projeto de um Processador Open Source em Bluespec Baseado no Processador Soft-core Nios II da Altera**. 95 p. Dissertação (Mestrado em Ciência da Computação) — Univeristy of São Paulo, São Paulo, 2014. Cited on page 45.

SOCIETY, A. M. **Weather Forecasting**. [S.l.], 2015. Available at: <<http://www.ametsoc.org/policy/weaforc.html>>. Accessed on: Sept. 27, 2015. Cited on page 25.

STOCKWELL, W. R.; KIRCHNER, F.; KUHN, M.; SEEFELD, S. A new mechanism for regional atmospheric chemistry modeling. **Journal of Geophysical Research: Atmospheres (1984–2012)**, Wiley Online Library, v. 102, n. D22, p. 25847–25879, 1997. Cited on page 31.

TAMULI, M.; DEBNATH, S.; RAY, A.; MAJUMDER, S. A review on jacobi iterative solver and its hardware based performance analysis. In: **proceedings of 1st International Conference on Power, Dielectric and Energy Management at NERIST**. [S.l.: s.n.], 2015. p. 10–11. Cited on page 85.

TEICH, J. Hardware/software codesign: The past, the present, and predicting the future. **Proceedings of the IEEE**, IEEE, v. 100, n. Special Centennial Issue, p. 1411–1430, 2012. Cited on page 46.

TSUCHIYAMA, R.; NAKAMURA, T.; IIZUKA, T.; ASAHARA, A.; SON, J.; MIKI, S. **The OpenCL Programming Book**. Fixstars, 2012. Available at: <<https://books.google.com.br/books?id=O86m1hJxA6QC>>. Cited 2 times on pages 37 and 40.

VAUGHAN, C. **Deep Thoughts on Deep Convection**. [S.l.], 2009. Available at: <<http://blogs.ei.columbia.edu/2009/03/01/deep-thoughts-on-deep-convection/>>. Accessed on: May 8, 2017. Cited on page 30.

VERWER, J. G.; SPEE, E. J.; BLOM, J. G.; HUNSDORFER, W. A second-order rosenbrock method applied to photochemical dispersion problems. **SIAM Journal on Scientific Computing**, SIAM, v. 20, n. 4, p. 1456–1480, 1999. Cited on page 31.

WANG, T.; WEI, P. Hardware efficient architectures of improved jacobi method to solve the eigen problem. In: **2010 2nd International Conference on Computer Engineering and Technology**. [S.l.: s.n.], 2010. v. 6, p. V6–22–V6–25. Cited on page 49.

WANNER, G.; HAIRER, E. Solving ordinary differential equations ii. **Stiff and Differential-Algebraic Problems**, 1991. Cited on page 31.

WARNER, T. T. **Numerical weather and climate prediction**. [S.l.]: Cambridge University Press, 2010. Cited 2 times on pages 25 and 26.

WILLIS, E. P.; HOOKE, W. H. Cleveland abbe and american meteorology, 1871-1901. **Bulletin of the American Meteorological Society**, v. 87, n. 3, p. 315–326, 2006. Cited on page 26.

WU, W.; SHAN, Y.; CHEN, X.; WANG, Y.; YANG, H. Fpga accelerated parallel sparse matrix factorization for circuit simulations. In: SPRINGER. **International Symposium on Applied Reconfigurable Computing**. [S.l.], 2011. p. 302–315. Cited on page 49.

YANG, C.; XUE, W.; FU, H.; YOU, H.; WANG, X.; AO, Y.; LIU, F.; GAN, L.; XU, P.; WANG, L. *et al.* 10m-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics. In: IEEE PRESS. **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. [S.l.], 2016. p. 6. Cited on page 52.

YARWOOD, G.; RAO, S.; YOCKE, M.; WHITTEN, G. Updates to the carbon bond chemical mechanism: Cb05. **Final report to the US EPA, RT-0400675**, v. 8, 2005. Cited on page 31.

ZHANG, H.; LINFORD, J. C.; SANDU, A.; SANDER, R. Chemical mechanism solvers in air quality models. **Atmosphere**, v. 2, n. 3, p. 510–532, 2011. ISSN 2073-4433. Available at: <<http://www.mdpi.com/2073-4433/2/3/510>>. Cited 2 times on pages 55 and 57.

ZHUO, L.; PRASANNA, V. K. Sparse matrix-vector multiplication on fpgas. In: **Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays**. New York, NY, USA: ACM, 2005. (FPGA '05), p. 63–74. ISBN 1-59593-029-9. Available at: <<http://doi.acm.org/10.1145/1046192.1046202>>. Cited on page 48.

_____. High-performance and parameterized matrix factorization on fpgas. In: IEEE. **Field Programmable Logic and Applications, 2006. FPL'06. International Conference on.** [S.l.], 2006. p. 1–6. Cited on page [49](#).

Appendix

INSTALLATION

In this Appendix, we describe the installation of BRAMS 5.2 and its libraries. Before compiling BRAMS and the necessary libraries, the user needs to install Intel Fortran Compiler¹ (ifort); our tests with gfortran failed.

After installing ifort, the user can proceed to compile the libraries² below. We provide useful links to libraries and BRAMS source code in Annex A.

- netcdf 4.1.3;
- hdf5 1.8.15;
- zlib 1.2.8;
- szip 2.1;
- mpich 3.1.4.

In this section, we consider the following directory structure for local installation³.

Main directory: home/<username>/brams

Libraries: home/<username>/brams/lib

Source code of BRAMS 5.2: home/<username>/brams/model

Each library demands its configuration, but the steps afterward are the same. The user needs to make, `make check` and `make install`. Do not ignore the `make check` command, the user must read the log file to find any particular problem.

- Compile Zlib.

¹ <https://software.intel.com/en-us/fortran-compilers>

² It is important to point out that it is necessary the same library versions cited in this section.

³ The user may choose another location.

```
1 $ CC=icc FC=ifort CPP=icpc ./configure --prefix=/home/<
  username>/brams/lib/zlib/1.2.8/
```

- Compile Szip.

```
1 $ CC=icc FC=ifort CXX=icpc F77=ifort ./configure --prefix=/
  home/<username>/brams/lib/szip/2.1/
```

- Compile Mpich.

```
1 $ CFLAGS=-O2 FFLAGS=-O2 CXXFLAGS=-O2 FCFLAGS=-O2 CC=icc FC=
  ifort F77=ifort CXX=icpc ./configure --disable-fast --
  prefix=/home/<username>/brams/lib/mpich/3.1.4
```

- Before the installation of Mpich you need to export the bin directory to bash file.

```
1 $ gedit ~/.bashrc
2 export PATH=$PATH:/home/<username>/brams/lib/mpich
  /3.1.4/bin
3 $ source ~/.bashrc
```

- Additionally you need to add your computer to the hosts list.

```
1 $ gedit /etc/hosts
2 127.0.1.1 <computer-name>
```

- Compile hdf5 and enable parallelism and fortran.

```
1 $ CFLAGS=-O2 FCFLAGS=-O2 CXXFLAGS=-O2 CC=/home/<username>/
  brams/lib/mpich/3.1.4/bin/mpicc FC=/home/<username>/
  brams/lib/mpich/3.1.4/bin/mpif90 CXX=/home/<username>/
  brams/lib/mpich/3.1.4/bin/mpicxx ./configure --enable-
  fortran --enable-parallel --prefix=/home/<username>/
  brams/lib/hdf5/1.8.15/
```

- Compile Netcdf with gcc and ifort, disable “netcdf-4” option.

```
1 $ FC=ifort F77=ifort F90=ifort CC=gcc ./configure --prefix
  =/home/<username>/brams/lib/netcdf/4.1.3/ --disable-
  shared --disable-netcdf-4 --enable-fortran
```

- Compile JULES and make some modifications.

- Backup the LIB directory of JULES;
- Rename makefile_ifort to makefile;

- Edit the variable BUILD=debug to BUILD=fast.

After installing all the required libraries, the user may proceed to BRAMS installation. BRAMS source code is available in Annex A.

- Go to /home/<username>/brams/model/build
 - Configure BRAMS;
- ```
1 $./configure --program-prefix=BRAMS --prefix=../install --
 enable-jules --with-chem=RELACS_TUV --with-aer=SIMPLE --
 with-fpcomp=/home/<username>/brams/lib/hdf5/1.8.15/bin/
 h5pfc --with-cpcomp=/home/<username>/brams/lib/hdf5
 /1.8.15/bin/h5pcc --with-fcomp=ifort --with-ccomp=icc --
 with-zlib=/home/<username>/brams/lib/zlib/1.2.8/
```

With BRAMS installed, the user may want to test it. CPTEC/INPE provides a test case; the user can also find this link in Annex A. The steps below refer to the installation of BRAMS's test case. The user must choose where it want to install the test case.

- Rename ANL, HIS and POSPROCESS to ANL.old, HIS.old and POSPROCESS.old;
- Create new directories with the respective names ANL, HIS and POSPROCESS;
- Rename RAMSIN to RAMSIN.bak;
- Make a copy of RAMSIN-5.1-meteo-only and rename it to RAMSIN;
- Download surface data and place it on datain directory;
- Edit RAMSIN file according to the following lines:
  - Rename all
 

```
1 /Users/saulofreitas/work to /home/<username>/run/bin/
 testcase_brams5.1/shared_datain/SURFACE_DATA
```
  - Rename CCATT = 0 to CCATT = 1;
  - Rename CHEMISTRY = -1 to CHEMISTRY = 4;
- Move the executable of BRAMS to <testcase\_home>;

For executing BRAMS, the user needs the following command:

```
1 mpirun -np N ./executable -f RAMSIN
```

Where *N* is the number of cores running in parallel BRAMS.



---

## OBSERVING THE RESULTS

---

BRAMS will write two outputs on POS directory. They are in Grads format. On Ubuntu it is possible to install Grads through apt-get:

```
1 $ sudo apt-get install grads
```

In other operating systems, it is necessary to download from Grads website. The link for Grads is also available in [Appendix A](#), download and install it according to the website instructions.

After installing Grads, go to POS directory. For each model output, there is a file with a description of extension .ctl and another extension .bin which contains the binary output. For opening them, use the following command:

```
1 $ grads -l
```

This command will display a prompt for the graphical output, and the terminal will return "ga->". With this prompt, the user can open a .ctl file, for example:

```
1 $ ga-> open light-A-2015-08-27-030000-g1.ctl
```

It will display something like this:

```
1 LON set to -64.9338 -59.8931
2 LAT set to -3.6793 1.35826
3 LEV set to 1000 1000
4 Time values set: 2015:8:27:0 2015:8:27:0
5 E set to 1 1
```

You can also check the variables in the file:

```
1 $ ga-> q file
```

The screen will display a variable list and other data. For plotting a variable, use the command “d” and the variable name as in the example below:

```
1 $ ga-> d tempc
```

For more information type help command. Now the user is able to see the results and compare with the test case provided by INPE/CPTEC. We used this test case for profiling the application.

---

## USEFUL LINKS

---

---

- Libraries:
  - [<ftp://ftp.unidata.ucar.edu/pub/netcdf/netcdf-4.1.3.tar.gz>](ftp://ftp.unidata.ucar.edu/pub/netcdf/netcdf-4.1.3.tar.gz)
  - [<http://www.hdfgroup.org/ftp/HDF5/current/src/hdf5-1.8.15-patch1.tar.gz>](http://www.hdfgroup.org/ftp/HDF5/current/src/hdf5-1.8.15-patch1.tar.gz)
  - [<http://zlib.net/zlib-1.2.8.tar.gz>](http://zlib.net/zlib-1.2.8.tar.gz)
  - [<http://www.hdfgroup.org/ftp/lib-external/zip/2.1/src/zip-2.1.tar.gz>](http://www.hdfgroup.org/ftp/lib-external/zip/2.1/src/zip-2.1.tar.gz)
  - [<http://www.mpich.org/static/downloads/3.1.4/mpich-3.1.4.tar.gz>](http://www.mpich.org/static/downloads/3.1.4/mpich-3.1.4.tar.gz)
- BRAMS:
  - [<ftp://ftp1.cptec.inpe.br/poluicao/BRAMS/src/BRAMS.tgz>](ftp://ftp1.cptec.inpe.br/poluicao/BRAMS/src/BRAMS.tgz)
- BRAMS testcase:
  - [<ftp://ftp1.cptec.inpe.br/poluicao/BRAMS/testcase/testcase\\_brams5.1.tgz>](ftp://ftp1.cptec.inpe.br/poluicao/BRAMS/testcase/testcase_brams5.1.tgz)
- Grads
  - [<http://www.iges.org/grads/>](http://www.iges.org/grads/)

