UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Applying Rosenbrock method for solving stiff ODEs raised from the chemical reactivity of the atmosphere through heterogeneous architectures based on FPGAs

Carlos Alberto Oliveira de Souza Junior

Tese de Doutorado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

ICMC USP
SÃO CARLOS

Carlos Alberto Oliveira de Souza Junior

Applying Rosenbrock method for solving stiff ODEs raised from the chemical reactivity of the atmosphere through heterogeneous architectures based on FPGAs

Thesis submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP, in accordance with the requirements of the Computer and Mathematical Sciences Graduate Program, for the degree of Doctor in Science. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Eduardo Marques
Co-advisor: Prof. Dr. Pedro Nuno Cruz Diniz

USP – São Carlos
May 2023

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

O48a

Oliveira de Souza Junior, Carlos Alberto
Applying Rosenbrock method for solving stiff ODEs raised from the chemical reactivity of the atmosphere through heterogeneous architectures based on FPGAs / Carlos Alberto Oliveira de Souza Junior; orientador Eduardo Marques; coorientador Pedro Nuno Cruz Diniz. -- São Carlos, 2023.
147 p.

Tese (Doutorado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2023.

1. Hardware. 2. FPGA. 3. OpenCL. 4. Codesign. 5. Heterogeneous-computing. I. Marques, Eduardo, orient. II. Nuno Cruz Diniz, Pedro, coorient. III. Título.

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

Carlos Alberto Oliveira de Souza Junior

Aplicando o método de Rosenbrock para resolver EDOs do tipo stiff oriundas da reatividade química da atmosfera através de arquiteturas heterogêneas baseadas em FPGAs

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

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Eduardo Marques
Coorientador: Prof. Dr. Pedro Nuno Cruz Diniz

USP – São Carlos
Maio de 2023

ACKNOWLEDGEMENTS

I dedicate this thesis to my grandparents, Ana and José, and sister Kelly for the support and the incentive during this period. That would have never been true without your hearing me out and supporting me during the long calls when I needed the most.

I appreciate everyone who collaborated on the development of this work and those who contributed somehow. First, I would like to thank my supervisor Eduardo Marques for the opportunity, efficient supervision, and the nights spent talking to me when I was anxious. I am grateful for the learning and support; they were much more than necessary. I immensely appreciate Professor Pedro Diniz and João Cardoso, who supervised me during my internship at FEUP-UPORTO. They were always available to read my work and provide insightful feedback.

I thank all the infrastructural and financial support for this research. Thank to the institutes and universities ICMC-USP and FEUP-UPORTO. To the "Fundação de Amparo à Pesquisa do Estado de São Paulo" (FAPESP) for the significant support given through processes no. 2017/14268-6 and 2019/07558-3. I also thank "Paderborn Center for Parallel Computing" and "Intel Labs Academic Compute Environment", for providing access to the Intel Hardware Accelerator Research Program (HARP) resources. The authors gratefully acknowledge the funding of this project by computing time supplied by the Paderborn Center for Parallel Computing (PC2). This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001

I would also like to thank Judimar, my cousin. She was responsible for financing my English studies. I hope you are proud to see this research written in English. That would have never been possible without you believing in my potential. I also could not forget my friends, Erinaldo, João Bispo, and Pedro Pinto, for understanding the importance of this work to me. Last and not least, my friends Lívia and Daniel were present, no matter my geographic location.

“People are really serious about software making its own hardware.”
(My adaptation from Alan Kay)

RESUMO

SOUZA JUNIOR, C. A. O. **Aplicando o método de Rosenbrock para resolver EDOs do tipo stiff oriundas da reatividade química da atmosfera através de arquiteturas heterogêneas baseadas em FPGAs**. 2023. 147 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2023.

Este trabalho foca na resolução de equações diferenciais ordinárias do tipo stiff através de métodos numéricos e com aplicação das técnicas de coprojeto de hardware/software. Estudos Anteriores mostraram que equações stiff requerem métodos implícitos para evitar passos muito curtos dos métodos explícitos. O problema é que estes métodos são baseados em conversões de sistemas não lineares para sistemas lineares, ou seja, é necessário resolver operações matriciais $Ax = b$. Durante o mestrado ficou claro que os sistemas lineares do CCATT-BRAMS exigem métodos diretos. No CCATT-BRAMS, isso é resolvido via método Rosenbrock que possui quatro estágios (somente o primeiro exige decomposição de matriz). Assim, é possível reaproveitar a decomposição para os próximos estágios do algoritmo para a resolução equações diferenciais ordinárias. O algoritmo de Rosenbrock foi dividido em duas partes, onde a primeira está relacionada com a resolução de sistemas lineares através de métodos diretos e a segunda com a modificação do Rosenbrock para aproveitar a arquitetura de FPGAs. Nossa revisão sistemática mostrou que há bem poucos trabalhos na literatura que exploram o paralelismo de equações diferenciais ordinárias em problemas de reatividade química para FPGAs. Nesta tese, provemos soluções para FPGA utilizando o Intel HLS OpenCL. Nossos resultados demonstram que a arquitetura de hardware gerada para o problema do CCATT-BRAMS é competitiva e que possui potencial para melhorar o desempenho e eficiência energética dessa aplicação tão importante para a previsão meterológica do Brasil.

Palavras-chave: Hardware, FPGA, OpenCL, Codesign, Computação Heterogênea.

ABSTRACT

SOUZA JUNIOR, C. A. O. **Applying Rosenbrock method for solving stiff ODEs raised from the chemical reactivity of the atmosphere through heterogeneous architectures based on FPGAs**. 2023. 147 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2023.

This work focuses on the solution of Stiff Ordinary Differential Equations through numerical methods applied to hardware/software codesign techniques. Previous studies reveal that such problems require implicit methods to avoid the very small timesteps of explicit methods. The problem is that implicit methods are based on the conversion from non-linear systems to a system of linear equations, which requires linear systems of the form $Ax = b$. During the author's master's thesis, it became clear that CCATT-BRAMS linear systems require direct methods. In CCATT-BRAMS, that is solved by Rosenbrock Method that includes 4 computational stages (only the first stage requires a matrix decomposition). In that manner, it is possible to reuse previous decompositions for the algorithm's subsequent stages to solve the ordinary differential equations. For that, we had to split Rosenbrock into two main tasks. The first relates to solving linear systems with direct methods and then modifying Rosenbrock to leverage the FPGA architecture. Our systematic review showed that very few works in the literature explore the parallelism of the stiff ordinary differential equations in chemical reactivity for FPGAs. In this thesis, we provide FPGA solutions based on Intel OpenCL HLS. Our results show that the generated hardware architecture is competitive and can improve the performance and power efficiency of such a critical application responsible for weather forecasting in Brazil.

Keywords: Hardware, FPGA, OpenCL, Codesign, Heterogeneous-computing.

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF SOURCE CODES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ATMET	ATmospheric, Meteorological, and Environmental Technologies
BDF	Backward Differentiation Formula
BRAMS	Brazilian developments on the Regional Atmospheric Modelling System
CB	Carbon Bond
CCATT	Coupled Chemistry Aerosol-Tracer Transport model
CPTEC	Center for Weather Forecasts and Climate Studies
CTM	Chemical Transport Model
DSL	Domain Specific Languages
DSP	Digital Signal Processor
EMIB	Embedded Multi-die Interconnect Bridge
FINEP	Financier of Studies and Projects
FPGA	Field Programmable Gate Array
GPL	General Purpose Languages
GPUs	Graphics Processing Units
HDL	Hardware Description Languages
HLS	High-Level Synthesis
IAG	Institute of Astronomy, Geophysics and Atmospheric Sciences
IME	Institute of Mathematics and Statistics
INPE	National Institute for Space Research
IP	Intellectual Property
JULES	Joint UK Land Environment Simulator
LCR	Reconfigurable Computing Laboratory
LE	Logical Elements
LUT	Look-Up Table
MIC	Many Integrated Cores
MIMD	Multiple Instruction Multiple Data
ODE	Ordinary Differential Equation
PBL	planetary Boundary Layer
PDE	Partial Differential Equation
RACM	Regional Atmospheric Chemistry Mechanism
RAMS	Regional Atmospheric Modeling System

RELACS	Regional Lumped Atmospheric Chemical Scheme
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SiP	System-in-Package
SOC	System-On-Chip
SOPC	System-On-a-Programmable-Chip
SPMD	Single Process Multiple Data
SRAM	Static Random Access Memory
USP	University of Sao Paulo
VHDL	VHSIC Hardware Description Language
VHSIC	Very High-Speed Integrated Circuit

CONTENTS

CHAPTER

1

INTRODUCTION

1.1 Context

Several engineering problems that rely on physical laws and relations can be modeled as differential equations. Such equations can be either Ordinary Differential Equation (ODE) or Partial Differential Equation (PDE) regarding the number of variables that the equation depends (KREYSZIG, 2010). Scientists have been modeling the laws of nature in mathematical expression to describe how nature behaves. Modeling such laws is crucial for fields such as engineering, physics, computer science, biology, medicine, environmental science, chemistry, and so forth. (KREYSZIG, 2010; TABAK, 2004).

Most of those laws of nature cannot be solved by analytic methods, which require numerical methods. In general, they are part of a software package becoming crucial tools for engineers. Although necessary, such methods impose the study of two principal variables: precision and performance, since numerical methods approximate the exact solutions.

We can now use and develop robust numerical packages with advanced computer architectures that provide fast and precise solutions. In this thesis, we focus on the chemical reaction problem, a system of Ordinary Differential Equations raised from the Coupled Chemistry Aerosol-Tracer Transport model (CCATT) Brazilian developments on the Regional Atmospheric Modelling System (BRAMS). This problem relies on a system of stiff ODEs using the Rosenbrock, organized as a series of 4 linear-solver steps, and each currently uses Sparse 1.3a, a sparse linear system solver library.

Sparse 1.3a is a package of subroutines in C for solving large sparse systems of linear equations based on LU decomposition. It provides efficient memory management through linked lists for the sparse structure. Besides, it also offers a Fortran interface, which allowed the engineers to couple this library to the numerical prediction of concentrations of chemical species in the atmosphere.

Predicting concentrations of chemical species requires a stiff ODE model. There is no exact definition of stiff, although intuitive for specialists (HAIRER; NØRSETT; WANNER, 1993). According to Chapra (2014), stiff equations have both fast and slow components in their solution. Usually, the components that vary rapidly die away quickly, and then the slowly varying components dominate the solution. Each chemical species concentration varies in different orders of magnitude, and such variation defines the stiffness of the chemical reaction equation.

Solving such stiff problems is computationally intensive, and it is usually solved by implicit or explicit methods. Curtiss and Hirschfelder (1952) were the first to conclude that stiff problems demand implicit methods such that the numerical solution is stable and converging at a reasonable computing time. According to Zhang *et al.* (2011) and Linford and Sandu (2009), chemical reactivity is intensive due to the implicit time-stepping algorithms, which require the solution of linear system equations. Our experiments also showed that matrix decomposition and solving $Ax = b$ is the most expensive operation of such methods.

Sartori (2014) explore two numerical implicit methods: (1) Rosenbrock, and (2) Backward Differentiation Formula (BDF). Rosenbrock is the current method used in production in BRAMS, and for BDF implementation, they used the LSODE library that contains a higher complexity order variation algorithm. The problem with LSODE is that they compute the problem for each grade point, which is different from how BRAMS works. BRAMS uses a group of points named blocks, which is potentially great for vector processors that require more data at the cost of a smaller step size for the block.

Their work does not explore parallelism or hardware implementation, and they are concerned with stability and numerical analysis, which is the core of numerical modeling, the most critical task in weather forecasting. It is also important to remark that the supervisor of this work is responsible for the CCATT-BRAMS implementation currently used in Brazil. Both implicit methods rely on expensive matrix decomposition and linear solver of the system of stiff ODEs. This thesis will focus on the current method implemented in BRAMS, the Rosenbrock. This method is already adapted for higher density of data, which is fundamental when considering heterogeneous computing. Another reason for choosing this algorithm is that both implicit methods have matrix decomposition as the most expensive operation due to converting nonlinear systems to a sequence of linear systems. Modeling an implicit algorithm for chemical reactivity is out of this thesis's scope, requiring advanced knowledge of atmospheric processes.

Regarding explicit methods, it is known in the literature that explicit methods are adequate for solving non-stiff differential equations or EDOs with a low degree of stiffness. That is, the chemical reactivity stiffness relies on the magnitude of the chosen species. For highly stiff problems, the step size would have to be tiny, consequently increasing the computational time.

Explicit algorithms for low stiff problems are highly parallel, and several studies apply such algorithms to GPUs. Stone and Davis (2013) develops the Runge-Kutta-Fehlberg algorithm,

an explicit algorithm of 4^{th} order (high-order), which is mandatory for explicit methods applied to stiff problems so they can be competitive with implicit methods. Their solution demands a parallel architecture because the sequential version is slower than the implicit methods, which directly affects the size of the integration step, as we mentioned earlier. The stiffer the equation, the more steps for converging.

Niemeyer and Sung (2014) also used an explicit method for stiff equations. They have implemented a low-order algorithm in GPU named Runge-Kutta-Chebyshev of 2^{nd} order. Their work concluded that highly stiff equations were up to $2.5\times$ slower than the parallel CPU implicit method implemented in VODE, and the solution is less precise. Since explicit methods are unsuitable for highly stiff equations, we are not considering the use of explicit methods in this thesis.

Besides choosing the appropriate numerical method for solving ODEs, we must also define the underlying architecture for the parallel solution. Since the 1970s, we have relied on the two most famous laws of computer science to improve our algorithms' performance: (1) Moore's law and (2) Dennard Scaling. The first states that the number of transistors would double every two years, and the second defines that as transistors get smaller and smaller, their power density stays constant (MOORE *et al.*, 1965; DENNARD *et al.*, 1974). Those laws increased the frequency and the number of cores per processor, improving performance.

As soon as the transistors reached a few dozen nanometers and increased their frequency, the energy efficiency started to get worse due to the leakage current of the transistors. That directly affects higher power density, which demands more heat dissipation from cooling systems. The higher the power density, the lower the performance gain. Too much power density leads to dark silicon, which describes the underutilization of the chip's resources (RAHMANI *et al.*, 2016). That phenomenon directly reflects the performance of many-core architectures.

Considering the current technology limitations, science and industry have been shifting to using accelerators that guarantee power efficiency besides the performance (LIU *et al.*, 2009; TSOI; LUK, 2010; THOMA *et al.*, 2015). The Field Programmable Gate Array (FPGA) is a technology used for high-performance computing and low power consumption. The modern FPGAs have included floating-point Digital Signal Processor (DSP), larger on-chip memory, and more adaptive logic resources that allowed the FPGAs to become close to the Graphics Processing Units (GPUs) consuming much less energy, therefore, better performance in GFLOPs/Watt (SANAULLAH; HERBORDT, 2018; MUSLIM *et al.*, 2017).

The use of FPGA requires deep knowledge of hardware different from the CPU and GPU that rely on software compilers. Unlike GPUs, FPGAs are not constrained to data parallelism. Its flexible and parallel nature allows the designer to implement any computation. Over the last year, Intel has invested in heterogeneous architectures containing a CPU and an FPGA in the

same die. One example of such a solution is the Atom E600C coupled to the Arria II GX FPGA[1], where such combination allows high-performance computing with low power consumption, also known as Ultra Low Power (ULP).

Intel has also cooperated with universities to improve such technologies through the Heterogeneous Architecture Research Platform (HARP) program. In HARP 2, they provided access to a chip containing a Xeon E5 v4 (CPU) coupled to Arria 10 (FPGA). Besides the previous advantages, this architecture improves the data communication overhead since both architectures share the same memory. For this thesis, this is one of the architectures used for our results.

This thesis relies on the Intel FPGA SDK for OpenCL since we use Intel's FPGAs. For a long time, FPGAs we being used only by experienced hardware engineers due to the difficulty of the Hardware Description Languages (HDL) (See experiment in Appendix A). As an approach to popularizing the FPGAs, the industry has been developing languages similar to the high-level languages widespread among programmers. Such languages are named High-Level Synthesis (HLS), and some of the most famous is Intel FPGA SDK for OpenCL, Xilinx Vitis HLS, and Java HLS (MAxJ compiler from Maxeler). Those languages rely on powerful compilers capable of generating an RTL from the HLS and then compiling for the FPGA.

1.2 Motivation and Objectives

According to (JUNIOR, 2015), using OpenCL is uncommon among meteorologists because converting the current models developed in Fortran 90 to OpenCL is difficult. Most of the solution for meteorological models in production relies on GPGPUs and Xeon Phi. None of the proposed solutions provide a portable source code, which has been abdicated since each architecture requires different optimizations for parallel strategies. Usually, those models for weather forecasting have a long life cycle, which is different from the hardware life cycle. That means any solution for such models must consider portability by including generic arguments that allow optimizations for different architectures.

As we mentioned before, GPUs suffer the most in dark silicon phenomena. That means we need to provide not only a fast solution but also a solution that is power efficient. That is crucial for BRAMS since the supercomputer (Tupã) was being threatened to be turned off due to the energy bill, which costs around one million dollars annually[2].

The main objective of this thesis is to provide a hardware/software codesign for the ODE implicit solver based on the Rosenbrock Method that should have a direct cause-effect on the power efficiency known on developed FPGA circuits. To achieve this goal, we have used

[1] https://www.eejournal.com/article/20101123-stellarton/

[2] https://g1.globo.com/sp/vale-do-paraiba-regiao/noticia/2021/06/15/diretor-do-inpe-diz-que-instituto-comprou-novo-equipamento-para-substituir-supercomputador-tupa.ghtml

OpenCL and Intel FPGA. Our current solution comprises data parallelism coupled with streaming processing. In the design, we mitigate the portability problems for the future architecture by including a set of parameters that allows the programmer to define the degree of parallelism.

1.2.1 Specific objectives

- Compare the architecture development to the literature regarding performance and type of parallelism whenever possible;
- Analyze the power consumption of the architecture
- Provide a parser from Fortran 90 to C-like for Fexchem and Jacobian. That parser also improves locality and removes sparsity, which drastically improves block RAM usage.

1.3 Thesis Structure

In Chapter 2, we describe the fundamental concepts necessary for this thesis. We start discussing Ordinary Differential Equations, stiff problems, CATT-BRAMS, FPGA, languages for hardware design, and heterogeneous computing. In Chapter 3, we contextualize our work in the literature and what has been studied in heterogeneous computing for solving stiff ODEs.

In Chapter 5, we show the 5 phases of the development of this thesis and how we designed the streaming solution. In Chapter 4, we describe the methodology we used from Fortran 90 to the OpenCL solution for the FPGA. In Chapter 6, we conclude our work and perform a final discussion on future work and contributions.

CHAPTER

2

FUNDAMENTAL CONCEPTS

In this chapter, we describe the main concepts related to our research. We define the mathematical concepts, which include the solvers for our problem. We also define our environment and the tools necessary for this research. Our last definition is our study case, BRAMS.

2.1 Ordinary Differential Equations

“Differential equations are extremely important in the history of mathematics and science, because the laws of nature are generally expressed in differential equations. Differential equations are how scientists describe and understand the world" (TABAK, 2004).

Differential equations can be either a PDE or ODE according to the number of variables. An ODE is a differential equation for a function of a single variable, and a PDE is a differential equation for a function of several variables (CHASNOV, 2016; SELICK, 2011).

In our thesis, we will work with the system of ordinary equations from chemical reactivity, a stiff problem. As a study case, we will solve the chem term from the continuity equation of the BRAMS. This term solves the chemical reactivity to measure air quality. For that, we provided an FPGA-friendly Rosenbrock implementation.

2.2 Continuity Equation

According to Jacob (1999), the continuity equation is the foundation for all atmospheric chemistry models. Such models support us in understanding how controlling processes (emissions, transport, chemistry, and deposition) can affect the concentration of species. That is a 3-D numerical model that simulates the variability of such processes in time and space; that model is called Chemical Transport Model (CTM) (JACOB, 2007).

CTM is a mathematical representation of the current knowledge about the processes in

atmospheric composition. By definition, they do not simulate atmospheric dynamics – they get meteorological information for input. Although, they do simulate atmospheric chemistry, as in the case of CCATT-BRAMS. The continuity equations from CTM can be either Eulerian or Lagrangian. CCATT-BRAMS derives the continuity equation in its Eulerian Form.

2.3 Stiff Equations

Stiff ODE can be an individual or a system of ODEs, with both fast and slow components in their solution. Usually, the components that vary rapidly die away quickly, and then the slowly varying components dominate the solution (CHAPRA, 2014).

In Equation 2.1, we present an example of a single stiff ODE:

$$y' = -100y, \quad t > 0, \quad y(0) = 1. \tag{2.1}$$

The exact solution is in Equation 2.2, where r is a constant.

$$\begin{aligned} \frac{\partial y}{\partial t} &= -100y \\ \frac{\partial y}{\partial t} + 100y = 0, \quad y = e^{rt}, \quad \frac{\partial y}{\partial t} &= re^{rt} \\ re^{rt} + 100e^{rt} &= 0 \\ e^{rt}(r+100) &= 0 \\ r + 100 &= 0 \\ r &= -100 \\ y(t) &= e^{-100t} \end{aligned} \tag{2.2}$$

In this example, the k^{th} derivative is $r^k e^{rt}$. As t increases, the derivative of r^k decays much slower than e^{-rt}. Since we have a term of this form, the error can be considerable if h is not small enough to offset this derivative. The larger r is, the smaller h must be to maintain accuracy (LAMBERS, 2010).

Requiring a tiny integration step (.e. small h) forces the time execution of the computation to become very slow. Moreover, most of the time, we must find the solution in a long-range (ŠÁTEK, 2011).

Due to the small integration times, systems of stiff differential equations cannot be solved by explicit methods. In this manner, we must use implicit methods for solving them. The following section shows some implicit solvers for stiff ODEs. In Figure 1, we show the plot from the stiff *van der Pol* equation, a visual example of stiffness.

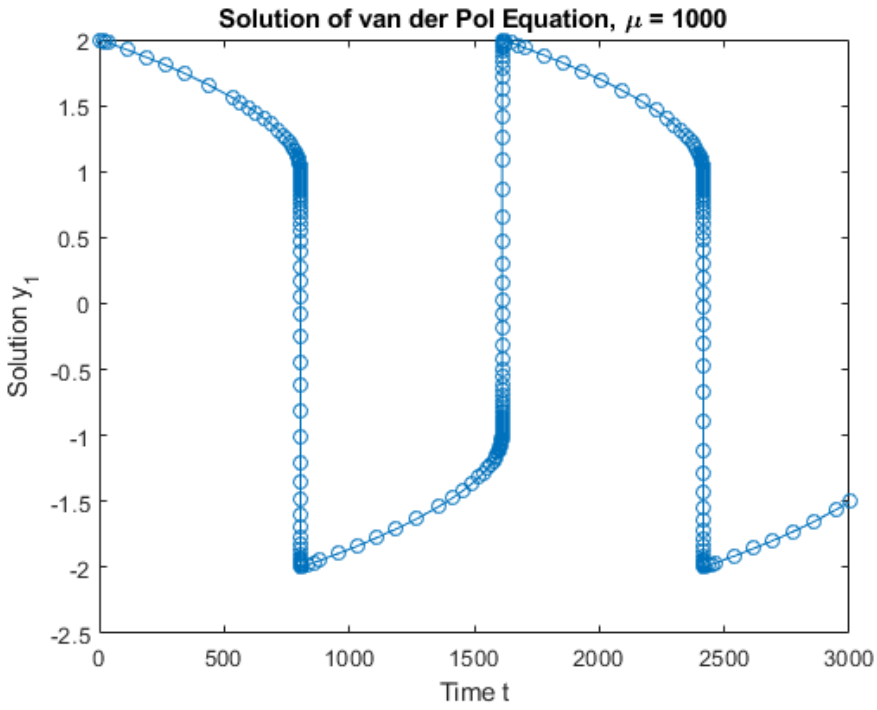

Figure 1 – van der Pol plot, figure from MathWorks (2018).

As we can see in the plot, some rapidly changing components require more points to decrease the error. Also, we have some slowly changing components, which can take advantage of bigger integration steps.

In general, stiff problems require the Jacobian matrix. Stiff solvers use the Jacobian matrix to estimate the local behavior of each ODE as the integration proceeds. Providing the Jacobian for efficiency and reliability is essential, especially in sparse systems. In Equation 2.3, we define the Jacobian matrix (MATHWORKS, 2018; SIMON; BLUME, 1994).

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix} \tag{2.3}$$

We must estimate numerically through finite differences if we do not provide the Jacobian matrix. In the context of BRAMS, every f is a function of the chemical reaction, and x is a chemical species.

2.3.1 Stiff problem

This project will consider the ODEs related to chemical reaction systems. We cannot provide a singular solver that can solve all the stiff problems since each problem has its stiffness rate and characteristics.

A chemical reaction is rearranging one or more molecules into a new substance. A chemical reaction can produce or lose energy, but molecular weights must maintain (HOWARD, 2009). In BRAMS, we have CCATT, the module responsible for air quality. CCATT solves the chemical reaction systems of ODEs by using a sequential algorithm, the Rosenbrock method.

2.4 BRAMS

BRAMS is a project initially developed by ATmospheric, Meteorological, and Environmental Technologies (ATMET), Institute of Mathematics and Statistics (IME)/University of Sao Paulo (USP) (IME/USP), Institute of Astronomy, Geophysics and Atmospheric Sciences (IAG)/USP and Center for Weather Forecasts and Climate Studies (CPTEC)/National Institute for Space Research (INPE) (CPTEC/INPE), and funded by Financier of Studies and Projects (FINEP) (INPE/CPTEC, 2022).

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 to large eddy simulations (LES) of the planetary boundary layer (LONGO *et al.*, 2013).

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

Three main models represent BRAMS: the tracer transport model, the chemical model (CCATT), and a surface model. BRAMS incorporates 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.

2.4.1 CCATT-BRAMS

CATT-BRAMS is an 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 (2.4).

$$\frac{\partial s}{\partial t} = \left(\frac{\partial s}{\partial t}\right)_{adv} + \left(\frac{\partial s}{\partial t}\right)_{PBL\ diff} + \left(\frac{\partial s}{\partial t}\right)_{deep\ conv} + \left(\frac{\partial s}{\partial t}\right)_{shallow\ conv} + \left(\frac{\partial s}{\partial t}\right)_{chem} + W + R + Q \tag{2.4}$$

"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); 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 are the energy transfer generated by the movement of liquid particles 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. This new model includes a gas phase chemical module, which solves the chem term in Equation 2.4. We show this module in Equation 2.5.

$$\left(\frac{\partial \rho k}{\partial t}\right)_{chem} = \left(\frac{d\rho k}{dt}\right) = P_k(\rho) - L_k(\rho), \tag{2.5}$$

The solution of this equation is the most expensive term of Equation 2.4. 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 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 2 illustrates the simulation of the CCATT-BRAMS system. The illustration represents the primary sub-grid scale processes involved in the trace gas and aerosol distributions.

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).

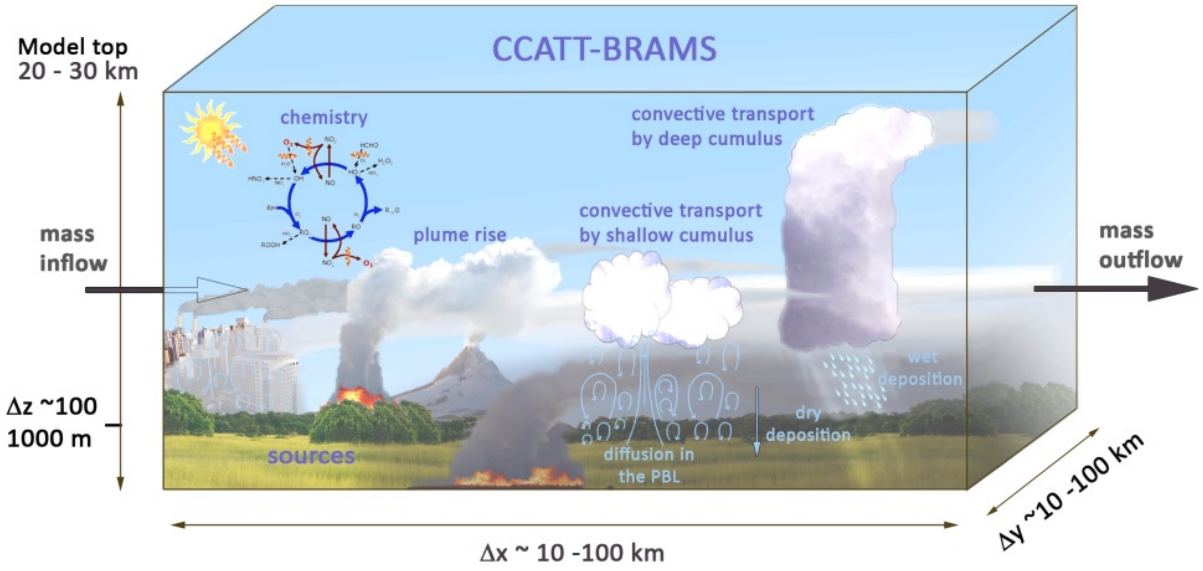

Figure 2 – Simulation of CCATT-BRAMS system, figure from (LONGO *et al.*, 2013).

Scientific projects frequently use the RACM mechanism due to the 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 Equation 2.5 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. This method adjusts the integration step 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 2.6.

$$\rho(t_0+\tau) = \rho(t_0) + \sum_{i=1}^{s} b_i K_i, \tag{2.6}$$

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 2.7a.

$$\begin{cases} K_i = \tau F(\rho_i) + \tau J(\rho(t_0)).\sum_{j=1}^{i} \gamma_{ij} K_j & \text{(2.7a)} \\ \rho_i = \rho(t_0) + \sum_{j=1}^{i-1} a_{ij} K_j & \text{(2.7b)} \\ F(\rho_i) = P(\rho_i) - L(\rho_i) & \text{(2.7c)} \end{cases}$$

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 2.8 is the most computing-intensive.

$$Ax = b \tag{2.8}$$

Where A is an $N \times N$ 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 Equation 2.7b by using Sparse1.3a (KUNDERT; SANGIOVANNI-VINCENTELLI, 1988). In Figure 3, we show the pseudo-algorithm for the Rosenbrock method with Sparse1.3a to solve each stage.

```
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(ρ) 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 ρ;
24                 Update production term F(ρ);
25                 Update b_i;
26                 foreach grad_point do
27                     Compute 3rd Rosenbrock method;
28                 Update matrix of concentrations ρ;
29                 Update production term F(ρ);
30                 Update b_i;
31                 foreach grad_point do
32                     Compute 4th Rosenbrock method;
33                 Update matrix of concentrations ρ;
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 3 – Rosenbrock Method.

According to Sartori (2014), the main references in the literature point to Rosenbrock and BDF as the most efficient implicit methods, mostly because they take advantage of sparse structures. Sandu *et al.* (1997) showed that Rosenbrock of 3rd with four stages (RODAS3) is similar to Rosenbrock of 4th order with six stages and requires less computational effort. RODAS3 is the current CCATT-BRAMS method and is the subject of study in this thesis.

Rosenbrock proposed the Rosenbrock Method in 1963 for solving stiff equations, also known as implicit Runge-Kutta. He developed a new class of single-step methods, where they substitute non-linear systems' solutions with a sequence of linear systems (much easier to implement). Such a method is used to solve Equation (2.7c), where the algorithm in Figure 3 can be represented as the Equation in Equation (2.9a).

$$\begin{cases} y_{n+1} = y_n + 2u_1 + u_3 + u_4 & \text{(2.9a)} \\ (\frac{2}{h}I - J)u_1 = f(y_n) & \text{(2.9b)} \\ (\frac{2}{h}I - J)u_2 = f(y_n) + \frac{4}{h}u_1 & \text{(2.9c)} \\ (\frac{2}{h}I - J)u_3 = f(y_n + 2u_1) + \frac{1}{h}u_1 - \frac{1}{h}u_2 & \text{(2.9d)} \\ (\frac{2}{h}I - J)u_4 = f(y_n + 2u_1 + u_3) + \frac{1}{h}u_1 - \frac{1}{h}u_2 - \frac{8}{2h}u_3 & \text{(2.9e)} \end{cases}$$

Where:

h = timestep
y_n = s(t) = the initial solution
J = Jacobian matrix
u = changing variable at the K^{th} stage

- For readability:
  - $(\frac{2}{h}I - J)$ equals to A matrix;
  - u is the current x solution;
  - $f(y_n)$ equals to b;
  - $f(y_n + z)$ requires the execution of the fexchem function because of the z variable;
- In that manner, we solve each stage as a system of linear equations of the Ax=b.

CCATT-BRAMS has run 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 time[1], as well as meteorological variables[2] (MOREIRA *et al.*, 2013).

[1] http://meioambiente.cptec.inpe.br/
[2] http://previsaonumerica.cptec.inpe.br/golMapWeb/DadosPages?id=CCattBrams

2.5 FPGA

An FPGA semiconductor device contains a two-dimension array of generic logic cells and programmable switches (CHU, 2011; MOORE ANDREW; WILSON, 2017). In 1985, Xilinx introduced the FPGA (BOBDA, 2007). In 2011, Amos, Lesea and Richter (2011) predicted that FPGAs would have around ten million logic cells, an accurate prediction for the current Stratix 10 GX 10M (Intel – around 10 million logic elements) and Virtex Ultrascale+ VU19P (Xilinx – around 9 million logic elements) FPGAs. In fig. 4, we show their prediction up to 2025.

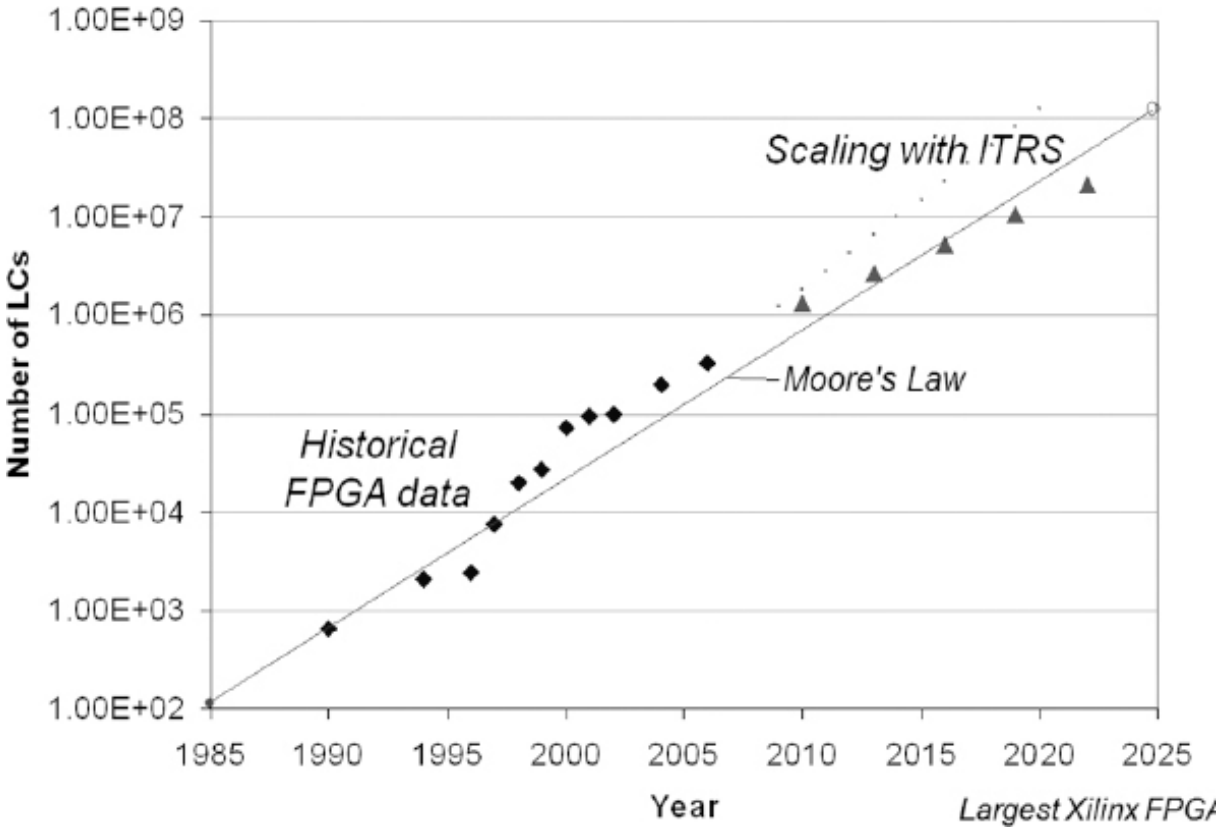

Figure 4 – Xilinx FPGA prediction, figure from (AMOS; LESEA; RICHTER, 2011).

With an FPGA, the programmer can define the behavior of the hardware after the manufacturing, which is why the name field is 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 Logical Elements (LE)s, the smallest unit of logic; usually, they are a Look-Up Table (LUT). Each LE can perform complex functions or basic logic as AND/OR. FPGAs also have configurable memory blocks, allowing 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 a commodity, which allows the programmer to reduce the cost and power of the design. These features allowed the programmers to build complex systems called System-On-a-Programmable-Chip (SOPC).

Generally, 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.

2.5.1 Architectures

Previously, our master's thesis used a Stratix V architecture. In this Ph.D. thesis, we focused on two updated FPGA architectures, and they are (1) Arria 10 from HARP 2 from Academic Compute Environment (ACE) and (2) Stratix 10 from Paderborn University. The first is a non-commercial variation of the Arria 10 coupled to a Xeon processor on the same die. We briefly discuss Agilex architecture because we intend to focus on it for future work.

2.5.1.1 Stratix V

Master's thesis results relied on the S5PH-Q-A7 Bittware board. That board connected the FPGA Stratix V to the CPU through a gen 3 x8, containing around 5.5 millions of gates (BITTWARE, 2015). Our Reconfigurable Computing Laboratory (LCR) bought this board for a project involving BRAMS and heterogeneous computing. This FPGA uses variable precision DSPs for implementing single and double precision for floating-point operations. The main reason for using such a Bittware solution was the Board Support Package for OpenCL, which allowed us to provide a heterogeneous solution for BRAMS. To the best of our knowledge, our work was the first to integrate FPGA and CPU in the same solution for solving CCATT-BRAMS linear system equations.

This architecture explicitly requires data movement between the CPU and FPGA, which is common in heterogeneous architectures. Regarding the floating-point operations, we noted that the final results tend to have a lower frequency and a higher number of logic elements used. Although it is the oldest architecture discussed here, this board is still a high-end FPGA with 52 MB of internal memory, the fastest resource to run out when solving matrix problems. The Bittware board is an excellent option for the parallel processing integer elements, but our results could have been more impressive when using the floating-point operations.

2.5.1.2 Arria 10

Arria 10 focuses on the HARP 2 architecture described on Faict, D'Hollander and Goossens (2019). On this architecture, three communication channels between the CPU (Xeon

E5-2600v4) and the FPGA (Arria 10 GX1150) are on the same die, which can achieve a theoretical peak of 30 Gb/s. We can also use OpenCL programming, which forces each bitstream to be split into two parts: (1) FPGA Interface Unit (FIU), and (2) Accelerator Functional Unit (AFU). That is a similar approach used on the Stratix V architecture, where part of the FPGA resources are used to implement the BSP.

In Figure 5, we show how the FPGA and CPU communicate with each other and how the FPGA splits the bitstream. It is essential to point out that FIU is fundamental for Arria 10 performance, which is the part that implements that Core-Cache Interface (CCI-P). This block abstracts away the low-level details from three channels to the programmer, which allows the use of shared virtual memory. Our results showed that this is critical for avoiding communication overhead.

This CCI-P also contains a cache coherent to the CPU cache and the DDR memory, which is critical to this architecture usage. From our experience, avoid using this cache whenever possible by inserting "volatile" over the global memory data. Most of the bitstream generation errors are avoided, and it increases the performance. We do not know the root of the cause, but for some kernels, the compiler cannot generate the circuit using this cache. When using it, we also noticed a drop in the circuit performance because when CCI-P does not contain the requested cache line, it fetches from the Last Level Cache (LLC) on the CPU, which results in higher latency. The worst case uses data from DRAM, which is fine if the programmer uses it only once during the production/consumer operations.

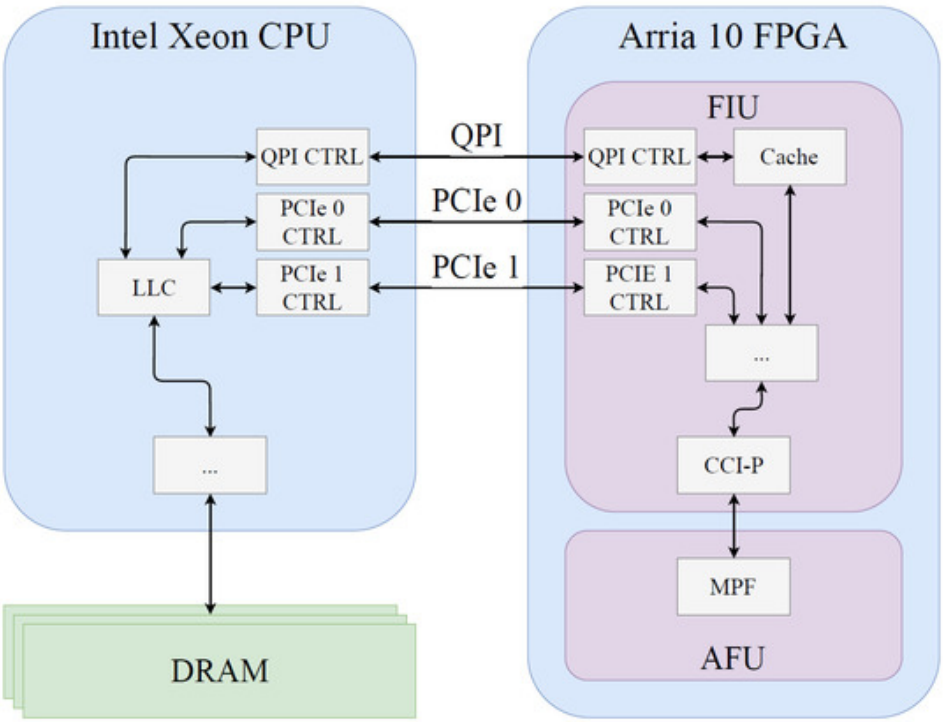

Figure 5 – Harp 2 architecture, figure from (FAICT; D'HOLLANDER; GOOSSENS, 2019).

Another advantage of this architecture is the floating-point DSP which improved hardware frequency, resource usage, and higher peak performance. According to our experience, this heterogeneous solution requires some optimizations when upgrading from Stratix V. Porting our previous Jacobi method source code without any modification to HARP 2 proved to be three times slower, even though communication improved in 50%. Those poor results are because the compiled solution to Arria 10 required more resources due to the memory replication applied to the registers, which lowered the circuit's frequency.

2.5.1.3 Stratix 10

The main difference of Stratix 10 is the new HyperFlex architecture. Intel added a register between ALMs throughout the core fabric, which improved critical paths by adding pipeline registers. That improvement also allowed a better usage of the ALMs since the designer does not need to sacrifice the logic functions because of their registers. This architecture is represented in Figure 6.

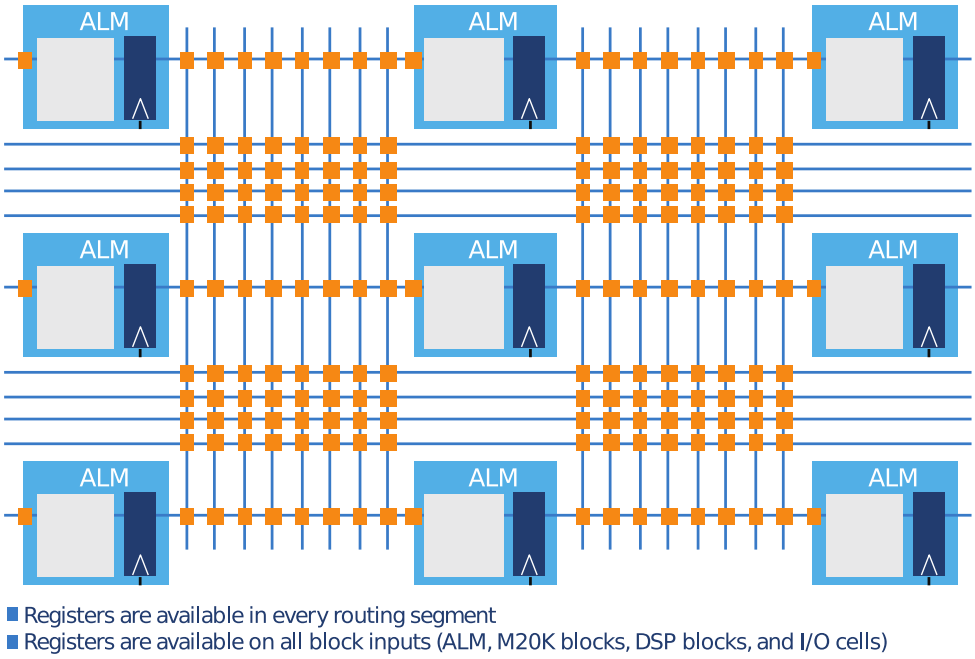

Figure 6 – Stratix 10 Hyperflex Architecture, figure from (HUTTON, 2022).

According to Hutton (2022), Intel's compiler can optimize timing after place and route without changing the design's routing. Although it seems fascinating, that came with some drawbacks like asynchronous resets must be converted to synchronous or removed because the hyper-registers (orange squares in Figure 6) are always synchronous. The architecture changed enough that Intel developed a new place-and-route algorithm for Stratix 10. They also implemented a new Fast Forward Compile tool that supports the designer on the RTL modification to improve performance (CHIU, 2021).

Intel also introduced 3D System-in-Package (SiP) technology in this generation. In short, this is an integration of different chiplets to the FPGA with different process nodes; that is, the external logic is decoupled from the core fabric of the FPGA. That is important to improve the time to market and avoid the expensive process of the 14 nm Stratix 10. The FPGA fabric and the chiplet communicate through the Embedded Multi-die Interconnect Bridge (EMIB). Regarding the Stratix GX 2800 that we used from Paderborn University, this is transparent to the designer. We noticed the amount of DSPs and embedded memory used as local memory for the OpenCL language.

For the theoretical peak performance of the Stratix 10 GX 2800, we have used the model proposed by Karp *et al.* (2021) because we did not find enough information for using our methodology in Section 5.3.3. According to their work, Stratix 10 can achieve a peak double precision performance of 500 Gflops/s. If we had used our methodology with the information mixed with Arria 10's, we would have around one teraflop of peak performance. We did not find ALM usage and flops per cycle for floating-point multiplication in double precision for Stratix 10, so we are using Karp's results. Regarding local memory, we have around 244 Mb on Stratix 10 compared to the 65 Mb of the Arria 10.

2.5.1.4 Agilex

Agilex is a direct upgrade from Stratix 10 mixed with Arria 10 from HARP 2. It used the second-generation Hyperflex architecture, more logic elements, higher memory bandwidth, double DSP count, PCIe gen 5 support, DDR5 support, and Compute Express Link (CXL) with 7 nm technology. This architecture comes in four flavors: D-series (midrange), F-series (midrange), I-series (high-end), and M-series (high-end) (INTEL, 2017). We focus on the M-series with High Bandwidth Memory (HBM) for this discussion.

Agilex has two game-changing features on its architecture: (1) HyperFlex gen 2 and (2) HBM. In the previous Stratix 10 architecture, the configuration RAM controls all the signals needed to go through a mux (see Figure 7). Now in the second generation (on the left), Intel has increased the speed of the signal bypass path. That optimization improves designs not tailor-made for the HyperFlex architecture, although the company states it is highly recommended. The second generation of HyperFlex did not fix the problem with asynchronous resets, which dropped the final circuit's performance. They also state that the second generation consumes up to 40% less energy than Stratix 10 (WON, 2022).

The second feature is related to the memory hierarchy of the Agilex-M, which contains HBM2e channels capable of 102 GB/s of peak memory performance. Figure 8 shows the memory hierarchy in the M-series. According to Velagapudi (2022) and Won (2022), HBM was possible due to the SiP connection introduced in the Stratix 10.

This new memory hierarchy allows the design teams to trade latency versus capacity. HBM has two orders of magnitude more capacity than on-chip memory and more than two orders

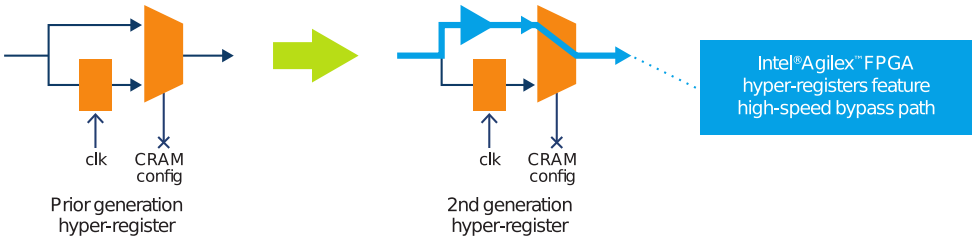

Figure 7 – Stratix 10 Hyperflex gen 2 Architecture, figure from (WON, 2022).

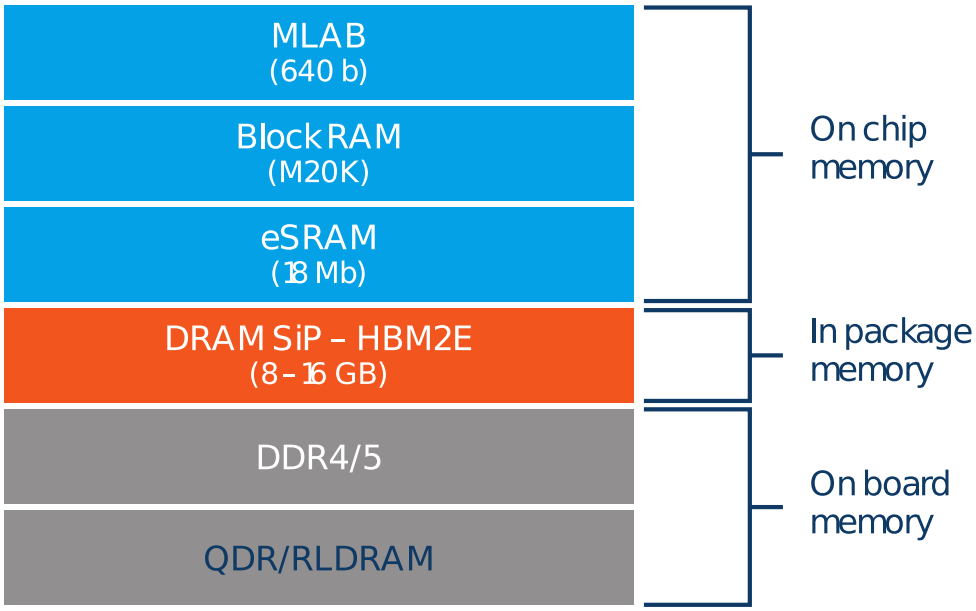

Figure 8 – Memory Hierarchy Agilex M-series, figure from (WON, 2022).

of magnitude of bandwidth compared to on-chip memory. For high-demanding applications, it is possible to use DDR5, 2× higher bandwidth than the DDR4 technology. Agilex also has a Network-On-Chip hard implemented for external memories, which avoids using FPGA resources as in the Arria 10 and Stratix 10 devices. Due to the 12,300 DSPs, this architecture can achieve 18.5 Teraflops in single-precision. Regarding the CXL, this is an upgrade from CCI-P from Arria 10 in HARP 2, where this technology is only enabled for a few Xeon series. As mentioned before, we could have had a better experience with CCI-P. We show a comparison table with the most critical elements of each FPGA architecture in Table 1.

Table 1 – Comparison among FPGA architectures used in this thesis.

	Arria 10	**Stratix 10**	**Agilex**
Logic elements	1,150,00	2,753,00	3,851,520
DSPs	1,518	5,760	9,375
Local Memory	65	244	311

2.5.2 Power Consumption

According to Seifoori *et al.* (2018), developing an exascale computer system requires about 3% of a typical nuclear plant's generating power, a prohibitive power consumption. That is also becoming challenging for FPGAs, especially the static power dissipation that is already over 50% of the total power consumption for 28 nm. Considering CPUs and GPUs that can reach up to 80%.

This thesis compares software implementation to FPGA implementations regarding performance and energy efficiency. Muslim *et al.* (2017) states that FPGA consumes much less energy because their control structure is hardwired; that is, it is unnecessary to fetch and decode the instructions. That is one of the reasons we are using FPGAs. GPU SIMD instructions require executing the source code twice whenever it finds a branch since the architecture cannot predict the behavior of the software. In FPGA, this branch is hard implemented, and the branch is chosen during the runtime execution. That also affects the energy-per-operation. Generally, a good balance of energy efficiency requires a hardware/software codesign, where the designer is responsible for mapping each application portion to the appropriate resource.

2.6 High-Level Synthesis

According to Nane *et al.* (2015), HLS tools have been developed for over two decades, and most are not maintained anymore. In this work, they present a plethora of languages that they distinguish into two groups: (1) Domain Specific Languages (DSL), and (2) General Purpose Languages (GPL). Most of the shown works use C or a subset of C as the input language and generate RTL in VHDL/Verilog/System Verilog.

One of the most famous HLS in the industry is OpenCL, an open standard for heterogeneous programming. The two biggest FPGA companies, Intel and Xilinx, implement their OpenCL compiler for hardware design. Xilinx HLs is primarily based on AutoESL, acquired by them in 2011. Later, it became Xilinx Vivado HLS and, finally, Xilinx Vitis HLS (XILINX, 2011).

Intel also has developed the Intel OpenCL Software Development Kit (SDK) for FPGAs, where they implement a subset of the OpenCL 1.2 version. Intel also provides specific optimizations for leveraging the FPGA resources, which are not portable for heterogeneous devices. Xilinx and Intel provide a set of techniques to enable pipelining, scaling, and efficient memory access (LAI *et al.*, 2021). The most recent HLS for Intel FPGAs is oneAPI, which has interoperability with OpenCL so developers can reuse existing source codes (INTEL, 2022). The main difference in the source code structure is that OneAPI describes the host and kernel into a single source file. This tool replaced Intel OpenCL SDK for FPGAs; because of this, we did not explore this framework for implementing our kernels. OneAPI promises the same benefits of the

OpenCL by accelerating the FPGA workloads effortlessly[3].

Intel provides some tutorials for migration from OpenCL to OneAPI, a language that works only with Intel devices. In both languages, the programmer can split its computation over single-task (pipeline parallelism) or ND-range (SIMD parallelism). OneAPI may be a problem for our current architecture since it does not support kernel volatile types, which is fundamental since CCI-P still presents some bugs.

According to (RODRIGUEZ-CANAL *et al.*, 2021) results, OneAPI is not optimized for the multiplication of tiny matrices, which is the case for CCATT-BRAMS. They also state that OneAPI is more efficient at using resources and performance when considering stencil computation.

2.7 Hardware Description Languages

2.7.1 VHDL

VHSIC Hardware Description Language (VHDL) is a hardware description language that describes the behavior of a system or electronic circuit. VHDL originated from Very High-Speed Integrated Circuit (VHSIC), an initiative of the security department from USA (PEDRONI, 2004).

It was the first IEEE language to become a standard by IEEE, called 1076. Its first version was in 87 and was later updated to VHDL 93. IEEE added a pattern, IEEE 1164, which adds the multivalue logic system. The designer can use VHDL for either circuit synthesis or simulation since not all descriptions are synthesizable (D'AMORE; CIRCUITOS, 2005).

- More abstraction to develop hardware, although we have system-level synthesis tools that provide much more abstraction;
- Libraries that implement common circuits for reuse;
- Vendor independent;
- Faster time-to-market.

2.7.2 Verilog and SystemVerilog

Verilog is a language similar to VHDL, although it is inspired by C. It is compact, and as VHDL, not all constructions are synthesizable (IEEE, 2005). System Verilog is an extension of Verilog that allows object-oriented programming, dynamical threads, communication among

[3] https://www.intel.com/content/www/us/en/developer/tools/oneapi/fpga.html#gs.lgb2y6

processes, and advanced features for formal verification of hardware (SUTHERLAND; DAVIDMANN; FLAKE, 2006). Modules represent the block projects, where each one contains input and output ports.

2.7.3 Bluespec SystemVerilog

Bluespec is a high-level synthesis tool that provides more abstractions for hardware designers, consequently speeding up the hardware design cycle (DAVE *et al.*, 2005). However, providing more abstraction limits the designer regarding the architecture integration since this task becomes part of the high-level- synthesis tool (MARTINEZ, 2017a). Figure 9 depicts the architecture of Bluespec.

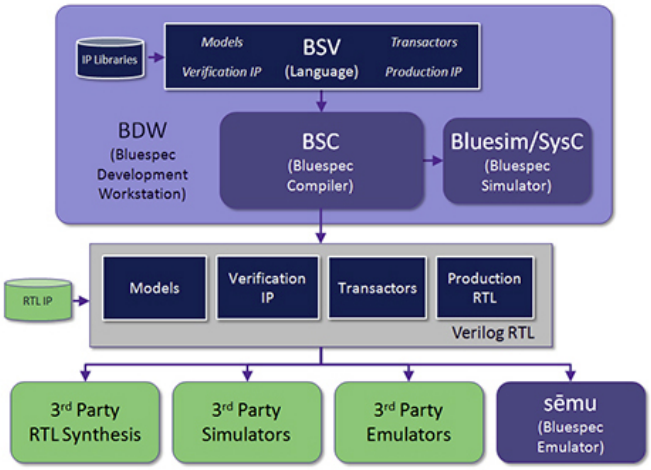

Figure 9 – Bluespec architecture, figure from Bluespec (2017).

Advantages of Bluespec language (MARTINEZ, 2017c):

- A hierarchical module similar to Verilog, which allows the designer to have control over the architecture;
- Atomic transactions. It automatically generates the logic control for concurrent access to shared resources;
- Atomic methods, which reduces the errors related to interface or integration problems;
- Source code fully synthesizable;
- Anticipated and fast simulation through Bluespec simulation tool.

2.8 OpenCL

Until 2004, programmers could improve software time execution by 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 10, we show the increase of clock rate and power (MURALIDHAR; BOROVICA-GAJIC; BUYYA, 2022).

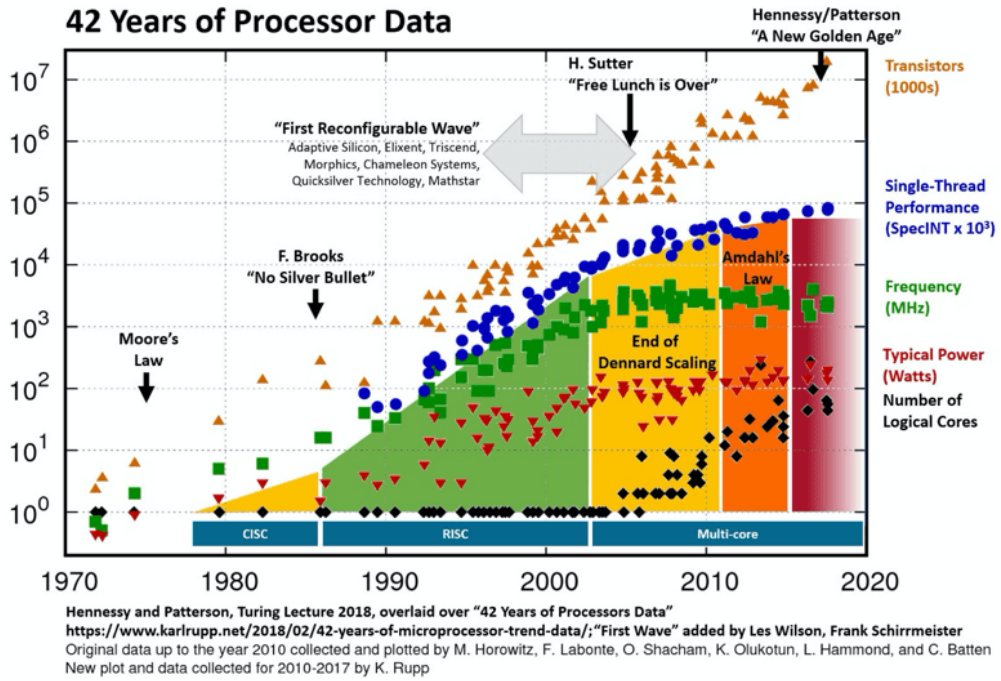

Figure 10 – Clock rate and power increase after 42 years of processor data, figure from (MURALIDHAR; BOROVICA-GAJIC; BUYYA, 2022).

From this point on, it was evident to the vendor that increasing the clock rate was not possible anymore. That forced the vendors to invest their money and efforts to change the 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 the CPU, it is possible to use parallel programming for accelerators; in this thesis, we consider accelerators for every non-CPU hardware. Shifting towards multicore technologies imposes a severe change in software development, especially if there is the 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 an extensive range of systems, from smartphones to supercomputers; this framework delivers much more portability than any other parallel programming standard (MUNSHI *et al.*, 2011).

2.8.1 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 the 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.

As the name says, the device 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 for implementing the kernels.

Memory objects maintain the necessary data (on the device) for the kernel's use. The API guarantees the order of memory transfers and kernel execution through the command queue. 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 11, it is possible to see how data structures interact. This picture represents OpenCL mapped to an FPGA device (green box). In this manner, the program resides inside the FPGA.

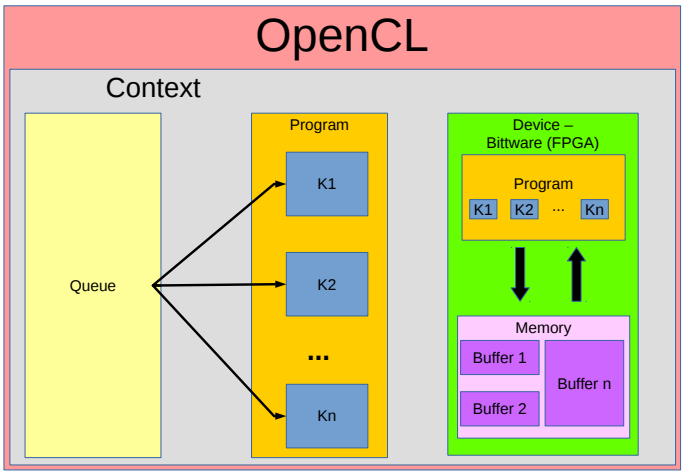

Figure 11 – OpenCL Data Structures – Consider the Program as a single data structure; we replicated it to make the understanding easier.

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 applying in OpenCL; defining which is better depends on the algorithm and the underlying hardware.

2.8.2 Data Parallelism

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

OpenCL API defines this programming model through 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.

This author believes 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, 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 of rows.

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 12.

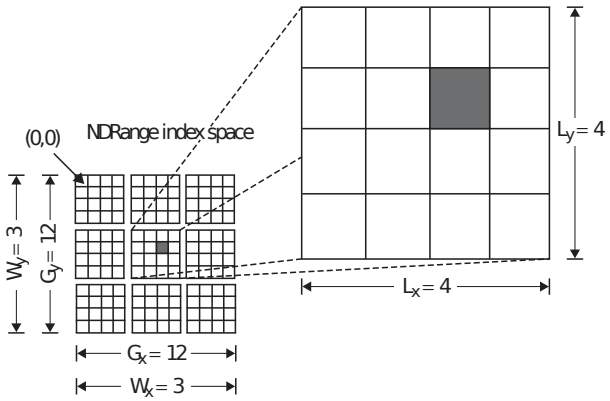

Figure 12 – An example of how the global IDs, local IDs, and work-group indices are related for a two-dimensional NDRange. For this figure, we have the following indices: the shaded block has a global ID of $(g_x, g_y) = (6,5)$, a work-group ID of $(w_x, w_y) = (1,1)$ plus a local ID of $(l_x, l_y) = (2,1)$, figure from (MUNSHI, 2009).

By using work-groups, OpenCL API imposes some restrictions on 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.

2.8.3 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 simultaneously in different pipeline stages; it is a crucial feature considering FPGA devices. Note that we did not mention GPUs; as we mentioned earlier, these devices are suitable for data parallelism due to the number of available cores.

2.9 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 master's thesis, we used Intel implementation for OpenCL in FPGAs.

Debugging is another critical leading factor that guarantees correct kernel functioning by emulating the CPU. By using the OpenCL standard, we could abstract away the FPGA design. This section presents key features of this standard applied to FPGAs from Intel. In Figure 13 we show OpenCL system implementation on the FPGA.

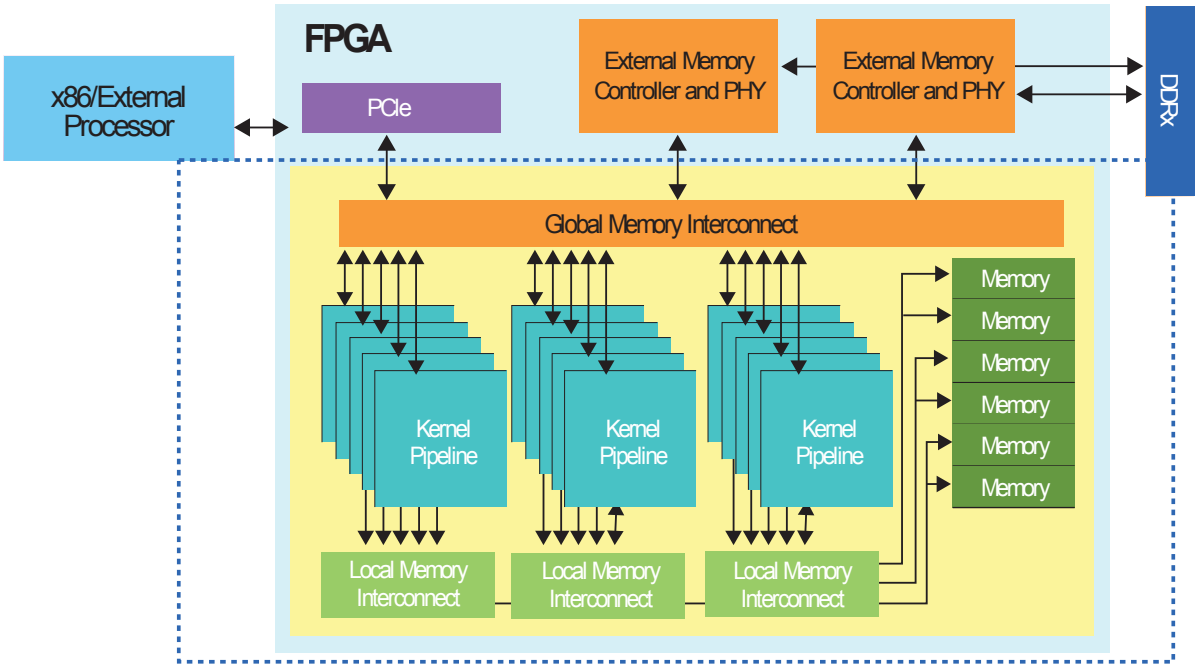

Figure 13 – Components from OpenCL system on Intel FPGAs, figure from (ALTERA, 2013).

This figure presents 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 14 shows the partitioning of the FPGA). The internal interface is critical to local memory (ALTERA, 2013; INTEL, 2016b). The external interface is necessary for accessing the Global Memory, which requires a global interconnect to manage the request from different pipelines; this global interconnect is also needed for the PCIe interface with the host.

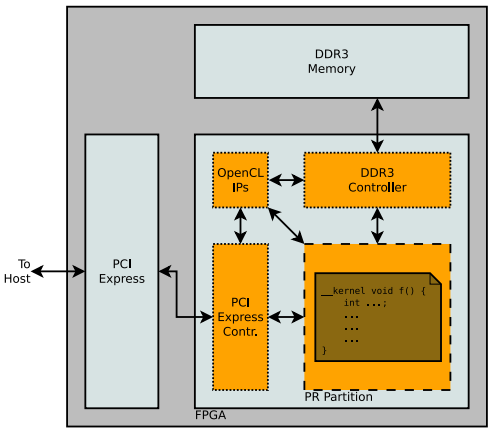

Figure 14 – Partitioning of the FPGA. PCIe, DDR3 controller and IPs are every project of OpenCL, so only the remaining is available for the kernels, figure granted by André Perina.

.

Unlike the GPU, where multiple cache levels exist, in FPGA, local memory requires M20K blocks spread over the board (INTEL, 2016a). In figure 15, we show a local memory with a single bank and three M20K blocks.

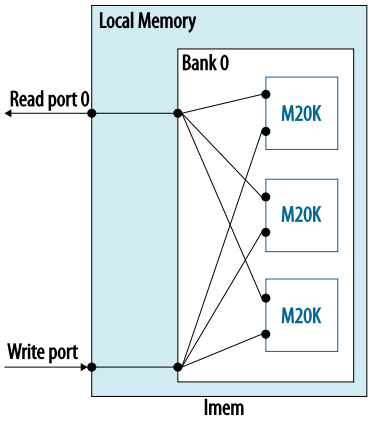

Figure 15 – Implementation of local memory with three M20K blocks, figure from (INTEL, 2016a).

Regarding private memory, Intel uses FPGA register to implement them. That is the fastest memory in the hierarchy, many of which are in the FPGA. The device can access these registers in parallel, which allows a much higher bandwidth than any other memory in OpenCL. According to our experiments, Intel infers registers for single variables or small arrays; a relatively big array requires local memory.

Intel performs several optimizations before generating the hardware, and Figure 16 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 AOCL Host Library. The kernel source code uses an offline kernel compiler (JANIK; TANG; KHALID, 2015), i.e., the programmer must compile the kernel separately from the host; this process may take hours to compile.

Compilation of the hardware is more complex than it seems. A C-language parser outputs an LLVM IR for each kernel (the 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 CDFG conversion takes place. The conversion is necessary to improve performance and 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 DDR3 memory. Finally, we can synthesize the complete system in Figure 13 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 kernel types on an FPGA.

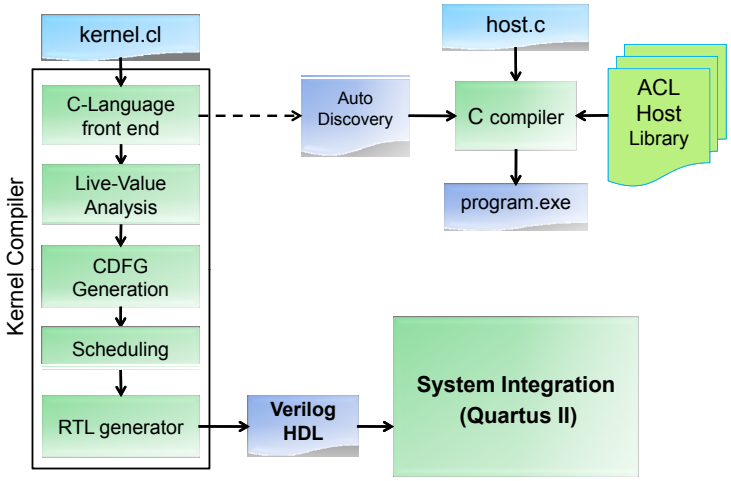

Figure 16 – Design flow with OpenCL, figure from (CZAJKOWSKI *et al.*, 2012b).

The main advantage of OpenCL over the traditional 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).

2.10 Codesign of Hardware/Sofware

A successful electronic system design requires using hardware/software codesign techniques (TEICH, 2012). Hardware/software codesign emerged in the 90s as a discipline; however, this task was already common among microprocessor companies. At the time, they needed to be conscious of the term codesign.

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 included an Intel microprocessor and a Stratix V FPGA (GUPTA, 2015).

Such technological evolution requires programmers to know about 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 definitions for hardware/software codesign.

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 17 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), and power density (processors can no longer increase the clock). Design complexity is much more challenging in hardware, design cost, and shrinking design schedules (time-to-market is reducing over the years, but software development can start even without a hardware platform).

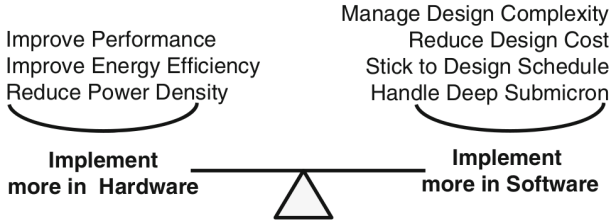

Figure 17 – Driving factors in hardware/software codesign, figure from (SCHAUMONT, 2012).

Partitioning or balancing is hard, and there is no magical solution. That requires expe-

rience; another important factor is the cost; it is often better to have a cheaper product than a fast one. Blickle, Teich and Thiele (1998) shows a theorem proving that determining a feasible allocation is an NP-Complete problem. In this work, they consider the problem of mapping a set of tasks to resources.

Some developed works in the literature consider 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 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. Our laboratory of reconfigurable hardware has experience in this field, as we can see in Martinez (2017b) and Pereira (2019).

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

CHAPTER

3

RELATED WORK

We have used ACM Digital Library, IEEE Digital Library, and Scopus for this chapter's systematic mapping. We have defined the following criteria for removing the studies: (1) duplicated study (removed with Mendeley tool), (2) non-English studies, (3) Computer Science as the only subarea (we want details on the algorithm and not numerical analysis over them). Some studies known previously were included based on experts' advice. Once we have all the papers, we read the titles and abstracts and perform another selection. Only then we read the entire paper and decide whether it is related.

Our systematic mapping made it clear that parallelism in stiff ODE solvers can be across the method or the system. Parallelism across the method means performing the stages in parallel, and parallelism across the system improves the linear system solver. All of the methods we found do not have dependency among the stages, which is possible depending on the problem they are modeling. In CCATT-BRAMS, the Rosenbrock method was modeled with dependency among the stages. We overcame that dependency by feeding a block of matrices to the Rosenbrock. In that manner, we process them in a streaming fashion through the pipeline. Most of the authors explore only one of the two types of parallelism. Several approaches also consider the approach where a processor contains a subset of ODEs and process them in parallel.

In this thesis, we focused exclusively on efforts that leverage FPGA-based accelerators. In our previous research (SOUZA; PEREIRA; MARQUES, 2017), we developed a parallel iterative method in FPGA based on the Jacobi method using a Stratix V FPGA with double precision. Since we are exploring both parallelism methods, we will split this systematic revision into two parts: (1) Parallel matrix decomposition and (2) Parallel stiff ODE solvers.

Our results showed that even parallel iterative methods are insufficient for CCATT-BRAMS since most of the computation must be redone several times. From those results, we evolved our work from iterative to direct methods for solving linear systems on the FPGA. In the literature, Kapre and DeHon (2009) replaced Sparse 1.3a with KLU. Both of those libraries

implement a variation of the LU Decomposition with double precision; the first one is the same used currently in CCATT-BRAMS. They realized that Sparse 1.3a was unsuitable for FPGA parallelism due to the frequent changes in the non-zero pattern of the matrices.

Their solution ranges from 300 to 1300 Mflop/s on a Xilinx Virtex-5, while the processor (Intel Core i7 965) achieved 6 to 500 Mflop/s. However, their work is computed on the FPGA, and the data/results are stored in DDR2 memory. In another work, Daga *et al.* (2004) implement an LU decomposition in double-precision in FPGA. They limited their solution to only non-singular matrices to avoid costly pivoting. The results are compared to a general-purpose processor, whose speeds range from 19 to 23. The entire solution is decoupled from the CPU, including the initial data.

In a similar FPGA-only design approach, Zhuo and Prasanna (2006), also report substantial performance improvements over their earlier versions making extensive use of block RAMs. In a genuinely heterogeneous design solution, Wu *et al.* (2011) propose a solution for sparse matrices, where the preprocessing is carried out in the CPU, and the factorization is in FPGA. Their simulated performance results rely on counting the factorization cycles and the processing element (PE) frequency but exclude the data communication overhead.

Ruan *et al.* (2013) present a similar approach to our previous research with Jacobi. They use a Java high-level synthesis (the MaxJ compiler) to implement the Jacobi method targeting a heterogeneous architecture fitted with Virtex-6 FPGA and a CPU. They also explored software parallelism with MPI and multithreading. Their heterogeneous solution is the fastest among their results, although it does not support matrices bigger than 200×200.

Regarding the implementation of QR decomposition, Parker, Mauer and Pritsker (2016) describe a pipelined implementation of the QR decomposition in single-precision targeting an Arria 10 FPGA and making heavy use of DSP Builder Tools. Their solution, with a peak performance rate of 78 Gflops, is not heterogeneous, as all the input data resides in FPGA block RAMs.

More recent work by Langhammer and Pasca (2018) describes an implementation of a QR decomposition based on the modified Gram-Schmidt in a core generator in C++, and the math operations were implemented in the DSP Builder Tools. Their work presents some modifications to the pipeline and the square-root operation using the reciprocal square root. Still, they use single-precision, which requires fewer cycles for divisor and reciprocal square-root operations. According to their results, they achieved three times the performance of (PARKER; MAUER; PRITSKER, 2016). Notably, they did not execute their application on the FPGA, as they used the frequency to simulate their peak performance.

In Ge *et al.* (2017), they also implement LU decomposition. According to their work, LU Decomposition is used for large-scale sparse linear equations, and its parallelism is difficult due to the data dependency of this decomposition. So they propose a cache-efficient architecture for

the Gilbert-Peierls algorithm, which relies on an elimination graph to avoid column dependency. They run their experiments on a Xilinx XC7K325T FPGA at 150 Mhz with single precision and compare to UMFPACK running on a CPU (Intel i5-3470 CPU). Their FPGA results range from $2\times$ to $10\times$ speedup compared to the CPU library. It is essential to remind that the matrices range size is from 1157 to 6833.

Cholesky is another matrix decomposition used in the literature. According to Sun, Liu and Zhou (2017), Cholesky decomposition is a particular form of the LU decomposition that deals with symmetric positive definite matrices (none of those properties are present in CCATT-BRAMS matrices). They process large sparse matrices (double precision) in their vector architecture on a Xilinx Virtex-7 FPGA VC709 evaluation board (xc7vx485t-2ffg1761). Their results are up to $2\times$ faster than the state-of-the-art for supernode algorithms (HSL_MA87).

Jiang and Raziei (2017) implements the Gauss-Jordan elimination algorithm, a direct method. That method requires one computation for each matrix since it can reuse previous decompositions. They provide a very low latency implementation running at 200 Mhz on a Xilinx Vertex 6 FPGA. Their results point to 5 microseconds for processing a 32×32 matrix. This work does not provide any information about data type, communication latency, or language used. Meng, Wakabayashi and Kuroda (2022) also provided a Gauss-Jordan implementation and used OpenCL HLS; their initial results point to 6.16 Gflops for matrices of 32×32.

Macintosh, Banks and Kelson (2019) provides a linear system solver based on truncated SPIKE that can simultaneously use FPGA, CPU, and GPU to solve tridiagonal matrices in single-precision floating-point. Their design uses a CPU Intel Xeon E5-1620 v4 at 3.50 GHz, a GPU NVIDIA M4000 8 GB GDDR5, and an FPGA Intel Arria 10 GX 8 GB DDR4. They process large matrices that range from 256 to 1280) $\times 10^6$, and their FPGA speedup is $1.7\times$ compared to the Intel MKL sdtsvb procedure. The FPGA requires $2.8\times$ and $20\times$ less energy than CPU and GPU implementations.

Our literature review found that matrix decomposition algorithms need to be more scalable for different matrix sizes. Most of the works in the literature are trying to solve linear systems with huge matrices to improve locality and use parallel units. LU decomposition is also likely unsuitable for small matrices, and most works restrict their solution to non-singular matrices to avoid costly pivoting. In this thesis, we adapted the work from Parker, Mauer and Pritsker (2016) to include a heterogeneous solution with OpenCL working in double precision. We published those results in "Exploration of FPGA-Based Hardware Designs for QR Decomposition for Solving Stiff ODE Numerical Methods Using the HARP Hybrid Architecture" (JUNIOR *et al.*, 2020).

The current state-of-the-art of matrix decomposition in GPU is QR householder, implemented in the MAGMA library. However, the library only executes in GPU if the matrix is bigger than 96×96. Otherwise, it is executed on the CPU. That is an optimization to avoid predominant communication in the final time execution (ANDERSON; SHEFFIELD; KEUTZER, 2012).

We are not considering solutions that perform poorly in the batched mode for the systematic mapping of the parallel matrix decomposition. That is methods optimized for tiny matrices, as for CCATT-BRAMS. Most famous libraries already implement such operations. Below we show the results found in the systematic mapping. We show the works' main features in Table 2

Table 2 – Main features of the linear system solvers in the literature.

	Iterative	Direct	Double	Memory type	Codesign
Jacobi (Souza, 2017)	x		x	Local/Global	x
QR (Souza, 2017)		x	x	Local/global	x
LU (Kapre 2009)		x	x	Local	
LU (Daga, 2004)		x	x	Local	
LU (Zhuo, 2006)		x	x	Local	
LU (Wu, 2011)		x		Local/Global	x
Jacobi (Ruan, 2013)	x			Local/Global	x
QR (Parker, 2016)		x		Local	
QR (Langhammer, 2018)		x			
LU (Ge, 2017)		x		Local/Global	x
Cholesky (Liu, 2017)		x	x	Local/Global	
Gauss-Jordan (Jiang, 2017)		x			
Gauss-Jordan (Meng, 2022)		x		Local/Global	
Truncated Spike (Macintosh, 2019)		x		Local/Global	

We explore the literature for methods that solve stiff differential equations for the second front. As mentioned, explicit algorithms are helpful for Ordinary Differential Equations with low stiffness. Equations with such variability in the answers are known as Stiff Equations. The more significant the variability, the stiffer the equation is. Explicit algorithms for low stiffed equations are highly parallel and have an advantage over CPUs. We show some works in this Chapter to confirm our statement (NIEMEYER; SUNG, 2014).

Due to the parallel architecture, most of the works in the literature implement such algorithms in GPUs. Such works are restricted to low-to-medium stiff problems since highly stiff problems do not perform well in GPUs. Such a problem happens due to thread divergence, which severely drops the performance of GPUs.

GPUs require Single Instruction Multiple Data (SIMD) applications to obtain performance. Each thread must perform the same operation, and stiff equations require a different step size for each equation. When we have thread divergence, the GPU must split the threads into groups with the same behavior and execute each group serially. We want to avoid this problem by using pipeline parallelism in this project. Our target architecture for that is the FPGAs, which over the years have been the most efficient regarding pipeline parallelism with the advantage of not suffering from thread divergence.

After refining our search string, we obtained 96 studies on solving Ordinary Differential Equations across the three digital libraries. We had to include one primary study with an expert's help. From the abstract, we selected 30 studies and read them all in full. Finally, the student

selected only 13 studies; most still needed to answer the questions, and two were unavailable. Table 3 presents the studies accepted.

Table 3 – Studies

Id	Title	Reference
S01	Numerical Results for a Parallel Linearly-Implicit Runge-Kutta Method	(BRUDER, 1997)
S02	Numerical solution of ODEs with distributed maple	(PETCU, 2001)
S03	The impact of different stiff ODE solvers in parallel simulation of diesel combustion	(BELARDINI *et al.*, 2005)
S04	A parallel algorithm to solve large stiff ODE systems on grid systems	(BAHI *et al.*, 2007)
S05	Generalized parallel algorithms for BVPs in ODEs	(KHALAF; ALNEMA, 2009)
S06	A parallel algorithm to solve large stiff ODE systems on grid systems	(BAHI *et al.*, 2009)
S07	Efficient SIMD solution of multiple systems of stiff IVPs	(KROSHKO; SPITERI, 2013)
S08	Parallel exponential Rosenbrock methods	(LUAN; OSTERMANN, 2016)
S09	10M-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics	(YANG *et al.*, 2016)
S10	GPU-accelerated solution of activated sludge model's system of ODEs with a high degree of stiffness	(ALIKHANIA; MASSOUDIEHB; BHOWMIKA, 2017)
S11	Accelerating finite-rate chemical kinetics with coprocessors: Comparing vectorization methods on GPUs, MICs, and CPUs	(STONE; ALFERMAN; NIEMEYER, 2018)
S12	Exploiting the multilevel parallelism and the problem structure in the numerical solution of stiff ODEs	(RUIZ; LOPERA; CARRILLO, 2002)
S13	Execution Behavior Analysis of Parallel Schemes for Implicit Solution Methods for ODEs	(KALINNIK; RAUBER, 2018)

Many authors are researching parallel solvers for stiff ODEs. According to our systematic mapping, we found 42 authors, and 12 of them are Chinese, where the fastest supercomputer of Top500 is located. Among of studies, 7 out of 13 are Europeans.

3.1 RQ1: What are the parallel methods (algorithms) used to solve stiff ordinary differential equations?

Through our systematic mapping, we could find four main parallel methods (algorithms) and their approaches to parallelism. Studies S01, S02, S03, S12, and S13 used Runge-Kutta methods, where all of them used Single Process Multiple Data (SPMD) approaches, that is, each process performs the same computation on different data. Although S12 exploits parallelism in two levels: task and data parallelism, they do not use any hardware that supports SIMD parallelism.

Studies S03, S05, and S10 used the Backward Differentiation Formula; interestingly, each author followed a different parallel approach. In S03, they used the SPMD approach; in S05, they used Multiple Instruction Multiple Data (MIMD) parallelism. In this paradigm, each process has a different computation over different data.

In S10, they used SIMD. This paradigm dictates that every core must compute the same thing over different data. Although SPMD and SIMD look the same, they are inherently different; in SPMD computation, each unit of computation has its data path, which is the opposite for SIMD devices.

We did not exclude any study with repeated authors, which is the case for S04 and S06 studies. They implemented the Waveform Relaxation algorithm with the SPMD approach. We wanted to check how they updated their algorithm and which insights were helpful for the improvement of precision or performance.

These studies propose a slower parallel algorithm requiring the minimum communication exchange in high-latency networks. Such studies showed the importance of communication and more than just parallelism in the algorithms.

Studies S07, S08, S09, and S11 used the Rosenbrock Method; this method is the same used in BRAMS, although it is sequential. We noted several approaches by the authors for this algorithm, each with its insight. In the S07 study, they only used a SIMD approach. In S08, they present an SPMD approach.

Studies S09 and S11 are attractive from the point of heterogeneous approaches. S09 uses SPMD with SIMD, and their processor is a Many Integrated Cores (MIC) with 260 cores, each core being an array processor. They used several techniques to make the most of the world's fastest MIC. The expert advised us that this study is the most important of them.

Finally, we have the S11 study. The authors present a Single Instruction Multiple Thread (SIMT) approach. Such a parallel paradigm is similar to SIMD, but each core can perform more than one task. They also used SIMD computation. They used GPU and MIC for both approaches. We summarize all the information in Table 4 and Table 5. In Table 5, RK stands for Runge-Kutta, WR is Waveform Relaxation and Ros is for Rosenbrock.

Table 4 – Parallelism approach.

	Parallelism			
Studies	**SPMD**	**SIMD**	**MIMD**	**SIMT**
S01	X			
S02	X			
S03	X			
S04	X			
S05			X	
S06	X			
S07		X		
S08	X			
S09	X	X		
S10		X		
S11				X
S12	X			
S13	X			

Table 5 – Algorithms.

	Algorithm			
Studies	**RK**	**BDF**	**WR**	**Ros**
S01	X			
S02	X			
S03	X	X		
S04			X	
S05		X		
S06			X	
S07				X
S08				X
S09				X
S10			X	
S11				X
S12	X			
S13	X			

3.2 RQ2: What is the precision of the parallel methods (algorithms) to solve stiff ordinary differential equations?

According to our systematic mapping, several studies must research the algorithms' precision and accuracy. Among them are S01, S02, S04, s06, S10, and S12; they do not even report the floating point precision.

Others reported that a poor study regarding precision was S05, S07, S08, S11, and S13. In S05, the author guarantees the error of the solution is low, although they had to increase the number of estimates of their method. In S07, they report only their algorithm's precision (double precision), and no error study was performed.

In S08, they compare the parallel version with the sequential version of the same algorithm and conclude that the parallel version presents similar results. In S11, they report that they used double precision and compared the results among their implementations; they do not report any error study. In S13, they perform a similar study.

Only two works performed studies regarding accuracy. In S03, they compare the solver result with the exact experimental result, concluding that their solver provides similar results for a low stiff problem.

In S09, the Chinese researchers present a detailed performance, precision, and accuracy study. They used double precision, proving that their algorithm is accurate for a high resolution, which until the year of this study, nobody had exceeded the resolution.

3.3 RQ3: What is the performance of each parallel method (algorithm) to solve stiff ordinary differential equations?

Our systematic mapping made it clear that parallelism can be across the method or the system. Most authors explore only one of the two types of parallelism (we explored both in this thesis). Several approaches also consider the approach where a processor contains a subset of ODEs and process them in parallel.

Studies S01, S02, and S08 explore the parallelism across the method. That is, they do not perform linear system operations in parallel. In S01 and S02, they could not perform better than the sequential version. In S08, they considered mildly stiff equations, which can provide more parallelism than highly stiff problems.

In S06, S07, and S10, they perform parallelism across the system. That is, they improved the linear system solver. They all provide good results, although S10 requires a matrix bigger than 128×128 to perform better than the CPU.

S09, S12, and S13 provide both method and system parallelism. In S09, they performed an excellent optimization to scale their parallelism to more than 10 million cores. In S12, they provide 5 to 8 times of speedup for dense systems.

Regarding sparse systems, they achieved $4\times$ of speedup. In S13, they had performance only with parallelism across the method. That likely happened because they split their problem with MPI processes, and no SIMD hardware was used.

The latter approach performs the computation by splitting the equations into subsets. We found that in S03 and S11. In S03, their algorithms performed better than BDF with low stiffness. In S11, they consider a highly stiff problem, but they need to consider communication time, which is unfair considering performance in heterogeneous systems.

3.4 Threats to Validity

This section presents some threads to the validity of our systematic mapping. Such threads can compromise some results, and they also showed us how to improve our Systematic Mapping.

Since only the Ph.D. student and the supervisor read and decided which paper was essential, we should expect some bias in the study selection and data extraction. The authors also struggled to decide whether the inclusion or exclusion criteria were impartial. The student tried to minimize the problems by exchanging ideas with Elisa Yumi Nakagawa, the expert in systematic mapping. However, she also needed help understanding the student's actual need

(since it is not her area) and defining what was necessary.

We know the possible exclusion of essential studies due to needing clarification on the titles or abstracts. The authors' studies may also need critical information that could return to our search string. We also did not search for specific conferences or journals in this topic area, which could bring many more related studies that could further improve our systematic mapping.

CHAPTER

4

METHODOLOGY

In this Chapter, we describe the methodology for this work. In Section 4.1, there is a description of BRAMS profiling studies that lead to the CCATT ODEs. Section 4.2 describes how we changed from Fortran 90 environment to C and then OpenCL for FPGAs. Section 4.3 briefly describes the phases of this work and how we solved the ODEs from CCATT-BRAMS. We also describe how measured performance and energy consumption.

4.1 BRAMS profiling

This thesis is an ongoing project from our master's thesis. In our first studies, we wanted to spot possible parallel algorithms that needed to be fully explored and responsible for some considerable time execution. For detecting such hot spots, we used gprof for profiling BRAMS into two configurations: (a) chemical module enabled and (b) with the chemical module disabled. We used those two because we already knew from an expert at INPE that checking every possibility of BRAMS is impossible because of the combinatorial explosion allowed by BRAMS's RAMSIM (configuration file).

Starting with the chemical module disabled, we noticed that the radiation function is responsible for 68% of the total execution. That is also the only function that exceeds 3% of execution. We have circled this function in Figure 18 generated with the gprof2dot tool. It is important to remind that this is a short version of the original call tree.

Considering the same methodology for the chemical module enabled, the Rosenbrock Method takes about 41% of total execution (see Figure 19). The second most compute-intense operation is spFactor, a sparse LU factorization method from the Sparse 1.3a library. That function is required for computing the linear systems raised from the ODEs and later solved by 4 stages in Rosenbrock. As mentioned before, this is the core of the chemical reaction and the most critical function for CCATT-BRAMS. Solving those linear systems is two times more

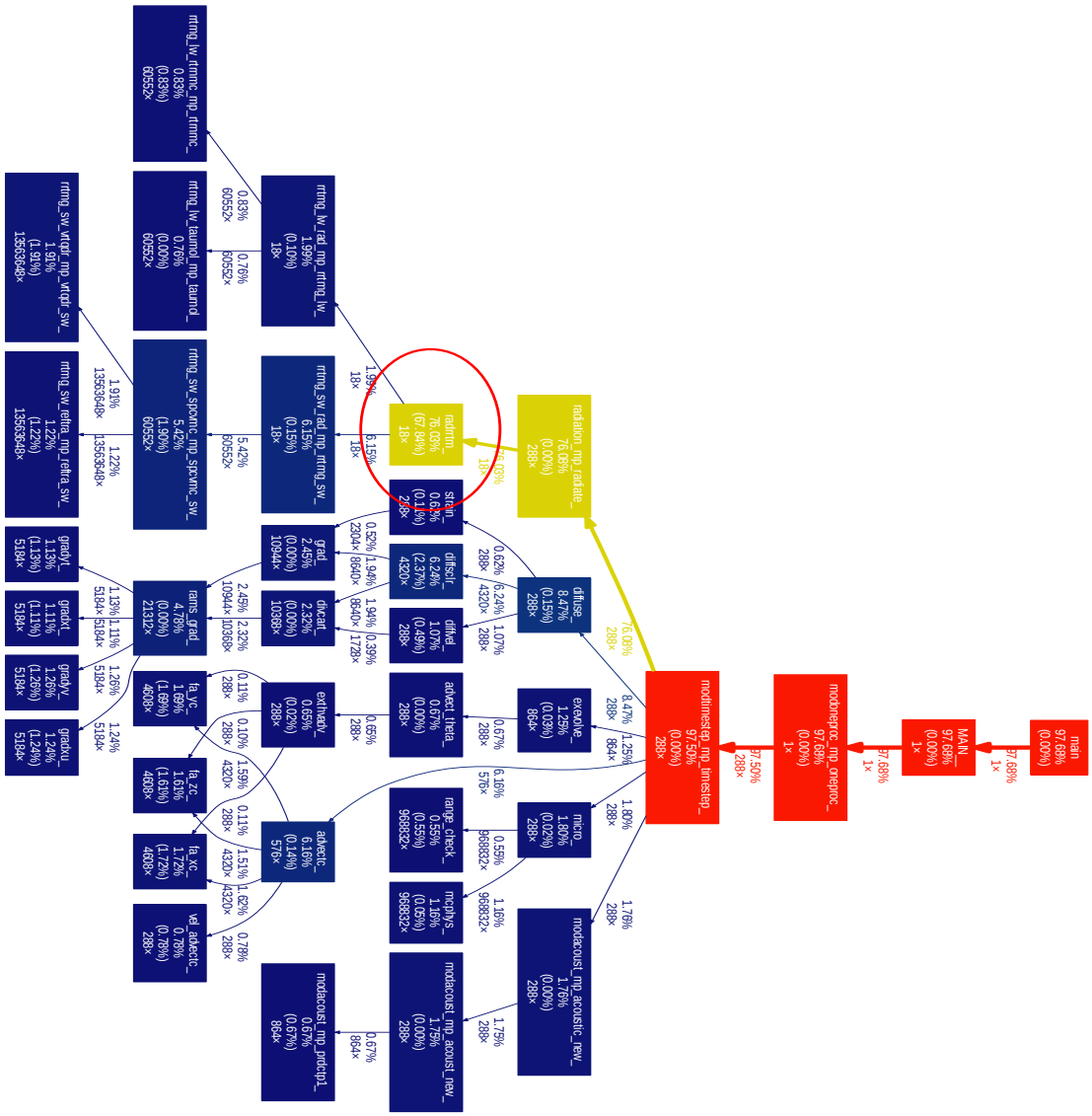

Figure 18 – Call Graph for BRAMS with chemical module disabled.

expensive than computing radiation and allows much more possibilities for parallel exploration.

4.2 Source Code Refactoring

BRAMS is a legacy source code implemented in Fortran 90 and C language. Our most challenging task was to port a heterogeneous solution that included OpenCL and Fortran 90, so we could execute the FPGA. The current Sparse 1.3a solution has a Fortran-ready interface to avoid extra effort for the programmer. Since performance and energy were among our constraints, we decided to avoid any automatic solution that included source-to-source compilation. Porting the sparse LU decomposition was not an option either, and as we mentioned in the related work, LU requires expensive pivoting operations.

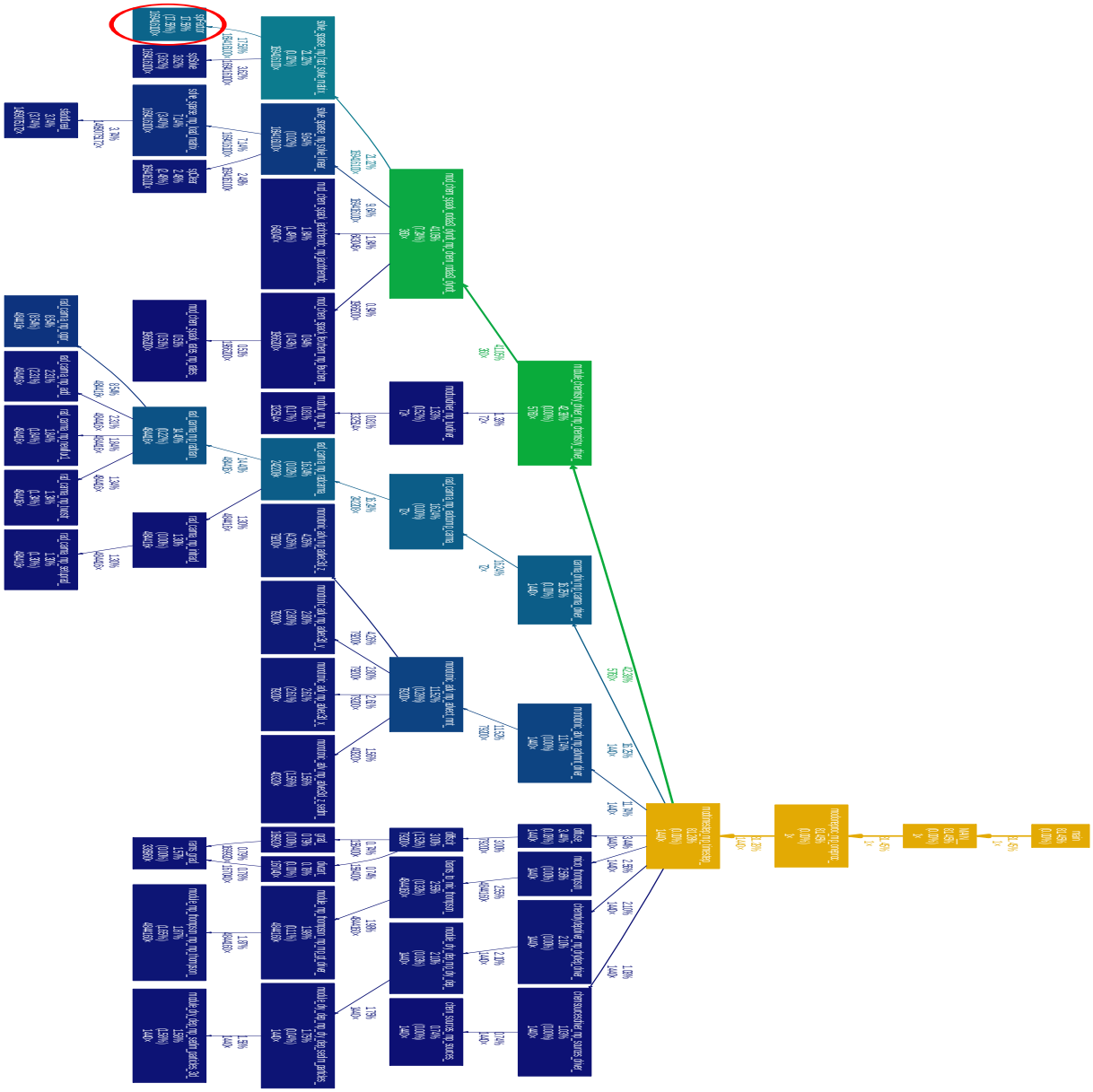

Figure 19 – Call Graph for BRAMS with chemical module disabled.

For solving CCATT-BRAMS with FPGA, we needed a specific solution for the target architecture. We already provide a portable source code, although we cannot guarantee its performance in other architectures. To guarantee the accuracy of the results, we used absolute error comparison. We executed the same function with CCATT-BRAMS functions and compared it to our equivalent solution in C or OpenCL. We were more concerned about the FPGA because we use compiler optimizations that affect the floating-point results by balancing the float operations. From previous experience, we are not considering NDRange kernels for this thesis. We wanted to provide a full pipelined version of the Rosenbrock solver.

The most time-demanding operation was compiling the OpenCL kernel for the FPGA. Depending on the algorithm's complexity, that task could take 12 hours. Thankfully Intel provided an environment for the compilation and execution of such kernels. The problem was

that we could not install BRAMS on their server due to the space limit (we need around 20Gb of data, it is possible to load only CCATT-BRAMS data) and administrator privilege. BRAMS requires several libraries that are unavailable for the user and need some root privileges.

To overcome such limitation, we executed BRAMS in a local server and stored only the necessary data for the Rosenbrock Method. We need the chemical production term, the Jacobian matrices, and their respective constants. This first source code selection resulted in around five thousand lines of code in Fortran 90 and C (see Annex C). For computing the production term and the Jacobian matrix, we had to rely on the tool developed by João Bispo at FEUP-UPORTO. He developed a parser for the mandatory data that improved memory locality, removed duplicated indices, and generated a C-like structure. Details on the implementation are in Chapter 5.

As an initial experiment, we developed this extracted software version of CCATT-BRAMS and executed it in HARP 2. From this point on, we first developed the algorithms in C, and after careful debugging, we implemented them in OpenCL. Rosenbrock solves an ODE by converting a non-linear problem to a linear problem with 4 stages. From the master's results, we knew that this problem required a matrix decomposition targeting FPGAs. We use an OpenCL adapted version of QR Decomposition developed by (PARKER; MAUER; PRITSKER, 2016).

During our research, we also noticed that implicit method like Rosenbrock relies on converting non-linear to linear problems, which is the most time-demanding operation. Sartori (2014) explores a refined solution named BDF implemented in LSODE. However, that solution does not allow the designer to explore vector parallelism like Rosenbrock because the library solves the problem for each grade point. Using a block of matrices like Rosenbrock resulted in higher execution time due to the smaller step size. This thesis focuses on implementing the Rosenbrock method targeting a pipelined solution for FPGA. Modeling an ODE solver from scratch is out of this thesis's scope, requiring deep numerical analysis and stability knowledge.

4.3 Work Phases

4.3.1 Phase 1

Our first work phase started in the master's thesis that implemented an iterative method for solving the linear system raised from the chemical reactivity of BRAMS. We also developed an OpenCL solution from scratch targetting a Stratix V (Bitware) connected to the CPU through PCIe and did not have DSPs specifically for floating-point operations. Back then, we explored both NDRange and task parallelism on the OpenCL. In the best scenario, communication was responsible for 10% of the total heterogeneous computation. This initial work concluded that even parallel iterative methods could not be a viable solution for ODE solvers that can take advantage of matrix decompositions.

4.3.2 Phase 2

That is the first implementation of this thesis, and we used our systematic mapping to reach the final solution that implements QR Gram-Schmidt targetting FPGAs. In this phase, we changed the Stratix V architecture to a newer heterogeneous solution provided by the Intel Hardware Accelerator Research Program (HARP 2), to which we had to submit a project and the University of Sao Paulo is one of the top universities around the globe that had access to it. In that new environment, we could share the cache and memory, communicate through three channels, and use a higher density of logic provided by Arria 10. Most of this thesis relies on this architecture for the results. This task resulted in a publication in (JUNIOR *et al.*, 2020). Our results showed that our matrix decomposition is still 5× slower than its software counterpart. Because of that, we wanted to explore more algorithms of the Rosenbrock in the FPGA. Our initial hypothesis was that we could overcome such a problem with more tasks in parallel and avoid communication with the CPU.

4.3.3 Phase 3

Moving more operations to the FPGA side required the results from the parser. That was necessary since Fortran 90 data structure is incompatible with OpenCL and allows more flexibility to the designs if they want to change the RELACS TUV mechanism. The parser can convert the chemical mechanisms implemented for CCATT-BRAMS to a C-like structure. We tried to explore a few parallel techniques, such as loop unrolling, but most of the time was spent fetching data. In this phase, we also analyzed every algorithm through the roofline model. That was important to guarantee the balance of operation between CPU and FPGA. Moving all those operations to the FPGA exhausted the local memory, pushing the performance down. Those results indicated that we needed to focus only on the 4 stages of the Rosenbrock and avoid memory-bound operations on the FPGA.

4.3.4 Phase 4

In this phase, we developed from scratch a streaming Rosenbrock capable of overcoming the dependency of the Rosenbrock stages. We created a pipeline over those stages, which is only possible because CCATT-BRAMS Rosenbrock is optimized to work with blocks of matrices. We use this feature for processing more than one matrix at a time. This solution requires communication with the CPU for each iteration of the Rosenbrock method.

That was necessary to avoid memory-bound operations like the Jacobian matrix generator. According to our results, this solution is still 7× slower than the sequential sparse LU with $-O3$. That is the first phase we measured the energy consumption and the energy per operation. We used a tool implemented in C by Intel named bdxpower. We throw the process during the execution of the 40 executions of our kernel, and it outputs an approximation of the energy spent

in Watts. For measuring the energy consumption of the software, we used Xeon's counters that are exposed through the pcm-power tool (INTEL, 2023a), where it splits the results into the energy spent in the CPU and the energy spent on the DRAM. We added both values for the total consumption of the software. To avoid any statistical error, we repeated the execution 40×.

4.3.5 Phase 5

Our previous solution exhausted the RAM blocks available on Arria 10. We used Stratix 10, an architecture with 4× more RAM blocks and twice the logic, as another attempt to improve performance. This architecture also uses a newer OpenCL version (21.4). That is one of the top architectures available in the market that we could access because we submitted a project to work on the Paderborn Center for Parallel Computing (PC^2) (CENTER, 2023).

Porting our previous source code to this new architecture did not work, and there was no warning on the compilation tool. The same source code caused a deadlock on the global memory access on the Stratix, and we believe that this was not happening on the HARP2 because there are three communication channels. We solved that by using mutex or preloading the global memory into local memory, which was only possible because Stratix 10 has a plethora of such resources. The final solution is around 183× slower than our best architecture with Arria 10, which is explained in Chapter 5. Regarding the energy results, we used the Nalla serial card monitor utility that comes with the Bittware 520N card, in which we measured the energy 100× during the 40 iterations of the same Rosenbrock algorithm.

CHAPTER

5

DEVELOPMENT

In this thesis, we are concerned with the chemical reactivity of CCATT-BRAMS, a system of stiff ODEs that models Brazil's air quality. 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), even for more minor scientific problems, we were able to replicate the intense computation roughly. Longo *et al.* (2013) uses the Rosenbrock method for changing from a nonlinear differential equation system to a linear algebraic increment in terms of K_i, where K stands for the chemical element. Such modification incurs in solving linear systems of the type $Ax = b$ through explicit or implicit methods.

In general, we solve stiff ODE problems in two classes of algorithms: implicit and explicit methods. Wanner and Hairer (1996) defines stiff equations as "problems for which explicit methods don't work"; they provide quite a narrow definition since explicit methods work with stiff equations. In our modest experience and opinion, a more accurate definition is "stiff equations are problems for which explicit methods can take a really long time to compute depending on the architecture".

Implicit solvers are the most efficient class for highly stiff ODEs, such as chemical kinetics in BRAMS. However, those implicit methods rely on complex algorithms to control step size, where the control flow imposes thread divergences. That is not a problem for CPU or FPGA architectures due to their large on-chip memory (SHI *et al.*, 2012). In the FPGA, both control flow paths are implemented, which removes the divergence at the cost of an area penalty. Opposite to them, GPUs suffer performance penalties from thread divergence because it forces serial execution (ZOHOURI; PODOBAS; MATSUOKA, 2018).

Since explicit methods usually have simple control flow and are more suitable for parallel environments, GPUs can offer better performance for this class of solvers for some balanced problems (STONE; DAVIS, 2013). We already have works in the literature that exploit

heterogeneous explicit/implicit methods, combining the strengths of both classes so they can use GPUs as accelerators for the moderate stiff side of the equations (SHI *et al.*, 2012).

For this work, we are using Rosenbrock implicit method because this algorithm avoids the computation of inverse for matrices and non-linear equations. Rosenbrock is also attractive for its step size control, which is performed automatically through embedded equations to estimate the local error. Such control relies on given values for ATOL and RTOL, absolute and relative tolerance. That is entirely problem dependent, and there is no one size fits all value. Since this is an implicit method, we can avoid computation using decomposition algorithms for the Jacobian matrix. In this manner, it is possible to compute one decomposition for the 4 stages of Rosenbrock.

In this chapter, we describe all the algorithms required for the Rosenbrock method to solve the chemical reactivity of BRAMS. We briefly describe our previous work in our master thesis and present the solutions for the problems we faced during the initial work that was continued in this Ph.D. thesis. Then we describe the 5 phases of this project that lead to the Rosenbrock Streaming algorithm, our main thesis contribution.

5.1 Phase 1 – Jacobian Iterative Method for Solving Linear Systems

During our master thesis, we wanted to test if a parallel iterative solver could outperform a sequential matrix decomposition for solving the linear systems for the chemical reactivity in CCATT-BRAMS. For that, we have implemented a Jacobian iterative solver in OpenCL by using the two approaches allowed by the language: (1) task and (2) NDRange. This section briefly shows our previous results and how that was the initial work for this thesis. In-depth detail of implementation and results can be found in our master thesis (SOUZA, 2017).

5.1.1 Jacobi Multi-threaded Dense

Our first implementation of Jacobian used a dense representation with the NDRange approach. We were already working with RELACS_TUV chemical mechanism with 47 species. Currently, INPE uses this mechanism for measuring the air quality in Brazil. CCATT-BRAMS can use any chemical mechanism the user provides; the limitation is related to the available computing resources. According to (LONGO *et al.*, 2013), RELACS can reasonably reproduce the results of the RACM, a chemical mechanism with 77 species.

Work-group size is the number of work items for each work-group. 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 second option is much more strict and does not let the compiler optimize the work-group size for

the problem. We tested both options, and the first generated the best hardware. Figure 20 depicts how work-groups and work-items map to our linear system.

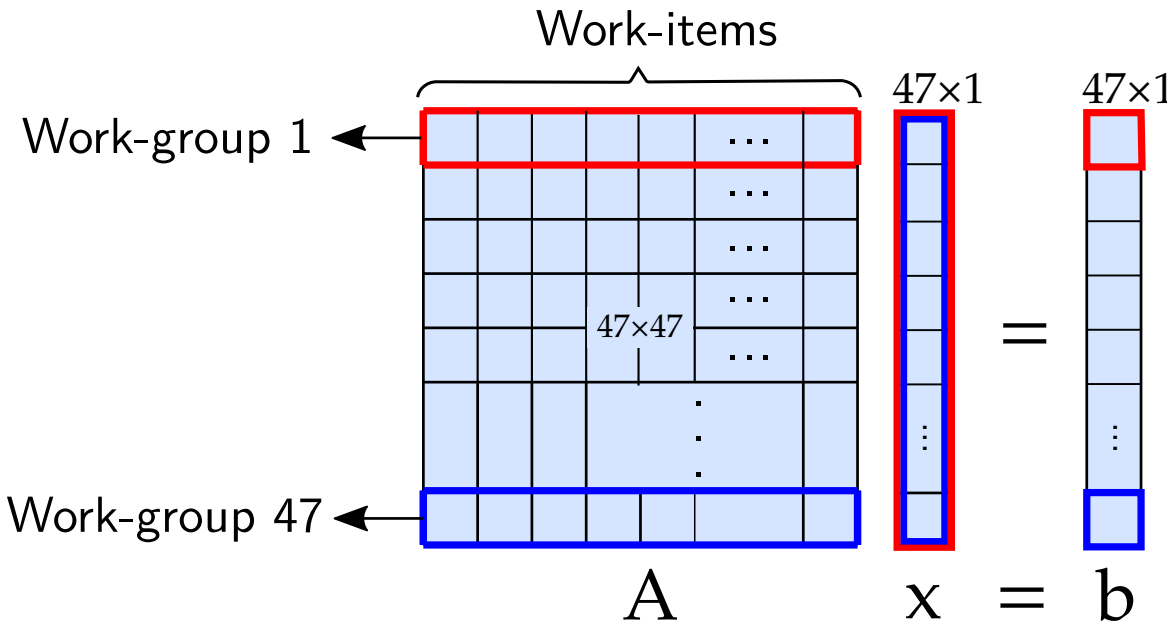

Figure 20 – Each color represents one work-group with 47 work-items. According to the Verilog, two work-groups are executing at the same time.

Each work group has a private memory space, preventing any NDRange application from synchronizing data among the work-groups. Since we needed to perform vector norm, we could either define a new kernel or perform the computation on the software side. Enqueue a new kernel is only worth it when there is a considerable amount of data for processing, which was not the case back then. Nowadays, we would solve this problem with OpenCL channels and avoid kernel communication with the CPU for this task.

Non surprisingly, this architecture presented terrible performance results. First, we computed a single matrix at a time and communicated with the CPU for every algorithm iteration. Besides, we were also using a dense representation for a matrix with only 10% density.

This communication exchange caused a severe drop in performance in BRAMS. Although we did not improve performance, we managed to maintain the results with a satisfactory error of $1.241371e-19$. On average, it takes 28ms to compute a single matrix. Considering that for a 24-hour simulation, we would need 44 days to get the results with this implementation.

Regarding energy consumption, we used PowerPlay Power Analyzer Tool from Quartus II (INTEL, 2023b); 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 the 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

5.1.2 Jacobi Multi-threaded Sparse

Our second approach implemented the same algorithm with a CSR sparse representation. According to our results, data movement was not improved since CSR uses more data structure, which directly translates to more transfers. A packed structure with the three arrays could improve the results, but it would still have CPU communication issues for every iteration. So, that was the root of the poor performance either.

We present the same results for this architecture in Table 8 and Table 9, as we can observe that the kernel execution improved, but the communication increased.

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

5.1.3 Jacobi Single-threaded Sparse

From the previous architectures, we knew that only changing the matrix representation was not enough. We must avoid communication with the CPU and perform the entire computation on the FPGA side. For that, we developed a single-threaded sparse version of the algorithm. We also boosted the performance using pinned memory, a type of non-paged memory.

According to our results, this architecture leads to the best performance and energy efficiency. That came at the cost of 15% more resources of the FPGA. This architecture takes around 19 hours for a 24-hour simulation. We summarize the results in Tab;e 10 and Table 11. From the results, it is possible to conclude that even the best performance possible is far from the ideal. We should try to provide a parallel direct method for solving the linear systems and avoid communication with the CPU.

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

5.2 Phase 2 – Direct Method for Solving Linear Systems

According to Golub and Loan (2013) and Peng (2013), there are two fundamental categories to solving linear systems: direct and iterative methods. In the previous section, we showed the results for the iterative method of Jacobi applied to BRAMS. We concluded that we needed to explore a direct method to reuse computation and avoid extra communication between CPU and FPGA. In this Section, we describe the current algorithm in BRAMS (LU decomposition) and the one that we developed to substitute it (QR decomposition).

5.2.1 Direct Method - LU

In theory, direct methods return the exact solution after a finite number of operations; in practice, this is impossible due to rounding errors. Lower Upper (LU) decomposition, Cholesky, Gaussian elimination, and QR decomposition are the main algorithms from this category.

Currently, BRAMS uses LU decomposition to solve 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 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. Its original purpose was for use in circuit simulators; it can also handle node and modified-node admittance matrices (KUNDERT; SANGIOVANNI-VINCENTELLI, 1988). This library manages the necessary memory for the sparse matrix by using linked-list representation; it also offers an interface for Fortran, making integration to BRAMS much simpler.

The current library used for LU decomposition in BRAMS is sequential. It requires an expensive update operation due to its pivoting, even though it uses a sparse format to decrease the amount of computing. So we needed a parallel direct method that could provide concurrency without pivoting, so we chose the QR decomposition (ANDERSON; BAI; DONGARRA, 1992).

5.2.2 QR Factorization

In this thesis, we explore implementing a direct method, the QR factorization, rather than the direct LU factorization method currently used in CCATT-BRAMS. Besides, we explored

two variants of the modified Gram-Schmidt QR (SINGH; PRASAD; BALSARA, 2007). The first implementation is a naive translation of the original algorithm to OpenCL without any improvement to take advantage of the underlying hardware; in the second, we implemented Intel's optimized version of QR (PARKER; MAUER; PRITSKER, 2016) for FPGAs. As a vehicle for our experiments, we rely on the heterogeneous HARP architecture from Intel using the OpenCL high-level programming language. We used an Arria-10 FPGA coupled to a Xeon CPU, allowing better communication and floating-point DSP blocks.

Since we do not have these algorithms implemented in BRAMS, we had to implement a software version in C language. Those implementations were necessary for the extrapolation of the results. We compare the FPGA hardware to these software versions regarding precision and performance. Over the following subsections, we describe both implementations of the QR.

5.2.3 The Original QR implementation

In this version, we implemented the QR factorization method based on Gram-Schmidt; we list it in Algorithm 1. We have implemented four variations, where the first is the baseline for the remaining three optimizations; they are as follows: (I) Straightforward translation from Algorithm 1 in C to OpenCL, without adding specific code for parallelism; (II) Insertion of shift registers to improve pipeline parallelism and remove data dependency on the multiply-accumulate inherited from the dot product operation; (III) Optimization to use local memory instead of global memory for performing the computation; (IV) Usage of the *-fp-relaxed* flag, which uses a balanced tree of floating operations by relaxing the order of the operations. Although useful for highly parallel hardware, it may incur numerical errors. We could not generate the fourth variation, as it did not fit on the Arria 10 FPGA.

Algorithm 1 – QR method without reordering (herein identified as QR).

```
1:  procedure QR(A,Q,R)
2:      Q ← A
3:      for k ← 1 to n do
4:          R(k,k) ← norm(Q(1 : m,k))
5:          for m ← 1 to n do
6:              Q(m,k) ← Q(m,k)/R(k,k)
7:          end for
8:          for j ← k+1 to n do
9:              R(k, j) ← dot(Q(1 : m,k),Q(1 : m, j))/R(k,k)
10:             for m ← 1 to n do
11:                 Q(m, j) ← (Q(m, j) − R(k, j)) ∗ Q(m,k)
12:             end for
13:         end for
14:     end for
15:     return Q,R                          ▷ QR Factorization of the Matrix A
16: end procedure
```

In Table 12, we show the timing results for each implementation. In Table 13, we present the results related to hardware resources. Those results suggest that the internal hardware generated for each version is similar, and most of the performance comes from the better usage of the memory hierarchy. Moving the computation to shift registers and local memory was enough to improve the (I) version in $15\times$.

Table 12 – Timing results for the original QR method without reordering.

	(I)	**(II)**	**(III)**
Send (μs)	18	16	17
Computation (μs)	66929	8038	4467
Receive (μs)	7	8	9
Total in HW (μs)	66954	8062	4493

Table 13 – Resource usage for the original QR method without reordering.

	(I)	**(II)**	**(III)**
Registers	196,011	301,857	290,391
Logic	126,903	184,110	171,421
DSPs	43	47	34
RAM blocks	538	810	731
Frequency (Mhz)	198.8	194.96	199.64

5.2.4 The QR based on Intel's implementation

Intel implemented an optimized version of the QR Grand-Schmidt for Arria 10 FPGA. They used DSP Builder tools and did not implement any codesign of their work. So we decided to implement the same algorithm in OpenCL and apply specific optimizations for the heterogeneous architecture HARP 2; we also had to change the algorithm's precision to double. Intel reordered the QR factorization to work into two functional groups; they computed the R terms before the Q terms. Lastly, they issued square root and divide operations as late as possible due to their high latency. We show this optimized version in the Algorithm 2.

We designed three variations of this algorithm, and they are as follows: (I) Insertion of shift registers and usage of local memory; (II) Compiler optimization with *-fp-relaxed* flag; (III) Compiler optimization with *-fpc* flag, that is responsible for avoiding rounding operations. Both compiler optimizations incur numerical errors, so we had to perform accuracy tests to guarantee the final results.

Table 14 shows the timing results for each version implemented. In the best scenario (III), we implemented a QR decomposition $4\times$ slower than the same algorithm implemented in the C language. According to our results, Intel implementation (Algorithm 2) does not map well to software. It is around 38% slower than Algorithm 1. In that manner, the best software version is based on Algorithm 1, and the best hardware version is based on Algorithm 2. Comparing both

Algorithm 2 – QR method with reordering.

```
1:  procedure QR(A,Q,R)
2:      Q ← A
3:      for k ← 1 to n do
4:          R2(k) ← dot(Q(1 : m,k))
5:          for j ← k+1 to n do
6:              Rn(k, j) ← dot(Q(1 : m,k),Q(1 : m, j)
7:          end for
8:          for j ← k+1 to n do
9:              for m ← 1 to n do
10:                 Q(m, j) ← Q(m, j) − (Rn(k, j)/R2(k)) ∗ Q(m,k)
11:             end for
12:         end for
13:     end for
14:
15:     for k ← 1 to n do
16:         R(k,k) ← sqrt(R2(k))
17:         for j ← k+1 to n do
18:             R(k, j) ← Rn(k, j)/R(k,k)
19:             for m ← 1 to n do
20:                 Q(m,k) ← Q(m,k)/R(k,k)
21:             end for
22:         end for
23:     end for
24:     return Q,R                                  ▷ QR Factorization of the Matrix A
25: end procedure
```

best implementations, we have a hardware QR decomposition $5\times$ slower than the best software version. Regarding the precision, we found an absolute error close to zero (0.002), which is enough for us to continue using compiler optimization on the floating point operations.

Table 14 – Timing results for the QR-based method.

	(I)	**(II)**	**(III)**
Send (μs)	17	15	18
Computation (μs)	5,658	1,176	950
Receive (μs)	8	9	10
Total in HW (μs)	5,683	1,200	978

5.3 Phase 3 – Memory analysis on the Rosenbrock Method

Our previous results with QR Decomposition showed we needed to explore more approaches to improve BRAMS' Chemical Reactivity. So we decided to port the Rosenbrock algorithm to the FPGA; we had to evaluate if the memory-bound operations inside the FPGA were better than the communication bandwidth required by a Hardware/Software Codesign. This

Table 15 – Resource usage for the QR-based method.

	(I)	(II)	(III)
Registers	349,291	472,231	287,786
Logic	216,907	263,752	166,794
DSPs	57	425	425
RAM blocks	2,145	707	642
Frequency (Mhz)	158.22	194.89	230

section describes the algorithms required by the chemical reactivity, their memory footprint, and a roofline model. We also describe the library developed by João Bispo at FEUP-UPORTO to support this thesis.

5.3.1 Parser for the Rosenbrock Indices

Porting Rosenbrock to FPGA demanded studying all the data structure around it and the necessary algorithms. We already implemented QR Factorization, but we needed to generate the data inside the FPGA to create a complete FPGA solution of the Rosenbrock. Most of the data was described in a table mode with several lines in Fortran 90, which was unsuitable for simple conversion from Fortran 90 to OpenCL.

We needed a fast approach to parse all of those data structures. For that, we used a customized parser created by João Bispo at FEUP-UPORTO. He used a library designed by their laboratory named specs-java-libs implemented in Java, where he used our Fortran 90 as the input and filtered each index of dratedc, jacobian, and fexchem. This parser, named Parser for the Rosenbrock Indices (PRI), generated the table of indices in C-like with a smaller memory footprint and better data locality since the Fortran version had a random data arrangement. Messing with indexation also forced him to remap the remaining algorithms to the new indexation so they could read the correct data. Those tables were essential to continue with our OpenCL implementation of those algorithms. We are not showing the source code due to its verbosity, but it is available in the GitHub of SPECS laboratory from the University of Porto[1].

5.3.2 Rates

Generating b (the production term) requires two algorithms: (1) Rates and (2) Fexchem. The rates function is responsible for computing reaction rates, and its output feeds the Fexchem function, which generates b. The rates function can be represented as a mathematical notation in Equation 5.1. Most structures are dense except for y indices (1Kb of data) extracted with regex

[1] https://github.com/specs-feup/specs-java-libs/tree/master/SpecsUtils/experiments-test/pt/up/fe/specs/util/jacobi

in Python. We show the source code representation in Source Code 1.

$$w(i) = rk(i) * y(j) * y(k) \quad \forall i = 1, 2, \ldots, reactions. \tag{5.1}$$

Source code 1 – Rates – Reaction Rates term in OpenCL

```
1  void fexchem_func(const int block_end, const int nreactions, const int nspecies,
2                    double local_rk[block_end][nreactions],
3                    double local_y[block_end][nspecies + 1],
4                    double local_dlr[block_end][nspecies]) {
5      double local_w[128];
6      const int y_indices[128][2] = {...};
7
8      for (unsigned int block = 0; block < 65; block++) {
9          for (unsigned int y = 0; y < 128; y++) {
10             local_w[y] = local_rk[block][y] * local_y[block][y_indices_fexchem[y][0]] *
11                          local_y[block][y_indices_fexchem[y][1]];
12         }
13     }
14 }
```

Some indices do not need the third element ($y(k)$), which was solved by padding 1 value to the last index of y. Whenever a third element is missing, we use this padded value for multiplication.

5.3.3 Fexchem

The Fexchem function is responsible for the chemical production term or b array from the $Ax = b$ equation. Generating this function also required the PRI to extract reaction rate indices, constants, index of constants, and the number of reactions rate per chemical species. Fexchem is represented in Equation 5.2 and listed in the Source code 2.

$$b(i) = \sum_{j=1}^{N} w(k) * const(index\_const(k)) \tag{5.2}$$

Where:

I = 1, 2, ..., species
N = Number of reaction rate per b_i
K = Index extracted by the customized parser.

The chemical production term (RELACS TUV) has a memory footprint of 8 Kb spread over four tables (that does not consider the compiler replication). To explore parallelism, we developed a mechanism that the programmer can define before compiling if they want data parallelism by changing two constant values.

Source code 2 – Fexchem – Chemical production term in OpenCL

```
void fexchem_func(const int block_end, const int nreactions, const int nspecies,
                  double local_rk[block_end][nreactions],
                  double local_y[block_end][nspecies + 1],
                  double local_dlr[block_end][nspecies]) {
    double local_w[128];
    const int y_indices[128][2] = {...};
    const unsigned int index_w[605] = {...};
    const double const_w[296] = {...};
    const unsigned int index_const_w[605] = {...};

    for (unsigned int block = 0; block < 65; block++) {
        for (unsigned int y = 0; y < 128; y++) { //rates
            local_w[y] = local_rk[block][y] * local_y[block][y_indices_fexchem[y][0]] *
                         local_y[block][y_indices_fexchem[y][1]];
        }

        unsigned int index_w_begin = 0, index_w_accum = 0;
        for (unsigned int y = 0; y < 47; y++) {
            unsigned int w_per_chem = index_w_per_chem[y] ;
            double shift_reg_chem[II_CYCLES_ACCUM + 1];

            #pragma unroll
            for (unsigned int i = 0; i < II_CYCLES_ACCUM + 1; i++) {
                shift_reg_chem[i] = 0;
            }

            for (unsigned int z = 0; z < ITERATIONS; z++) {
                double accum_chem = 0;

                #pragma unroll
                for (unsigned int i = 0; i < FACTOR; i++) {//data-parallelism
                    unsigned int index = z * FACTOR + i;

                    accum_chem += ((index < w_per_chem) ? local_w[index_w[index_w_begin]]
                                  * const_w[index_const_w[index_w_begin]] : 0);
                    index_w_begin++;
                }
                shift_reg_chem[II_CYCLES_ACCUM] = shift_reg_chem[0] + accum_chem;

                #pragma unroll
                for (unsigned int i = 0; i < II_CYCLES_ACCUM; i++) {
                    shift_reg_chem[i] = shift_reg_chem[i + 1];
                }
            }
            double sum_chem = 0;

            #pragma unroll
            for (unsigned int i = 0; i < II_CYCLES_ACCUM + 1; i++) {
                sum_chem += shift_reg_chem[i];
            }

            local_dlr[block][y & 0x2E] = sum_chem;

            index_w_accum += w_per_chem;
            index_w_begin = index_w_accum;
        }
    }
}
```

Although fexchem allows some degree of parallelism, it also incurs heavy memory-bound operations at fetching indices, multiplying, and accumulating over matrices. One of the best practices proposed by Intel is defining the arrangement of the local memory to avoid memory replication and loss of performance. It is encouraged that the programmer should provide the best banking possible according to their needs. The compiler can also do this automatically, but they do not guarantee the best performance.

When we provided the local memory arrangement, we achieved a slight improvement of 10% in the number of block RAMs at the cost of running 20%. Improving block RAM utilization increased the Initiation Interval (II) at lines 12, 18, and 27 in Source Code 2. We used the roofline model to analyze fexchem (OFENBECK *et al.*, 2014) to avoid unnecessary optimizations that will not improve performance. We show the results in Table 16.

Roofline presentation for the fexchem is represented in Figure 21. Due to the flexible nature of the FPGA, we will discuss whether it is possible to use all the available DSPs for double multiplications, the most common operation in the Rosenbrock algorithm. According to (INTEL, 2021), each double multiplication requires 4 DSPs, 312 ALMs, and provides a peak frequency of 288.35 Mhz, where each DSP can perform 2 flops per clock cycle. Our FPGA (Arria 10 GX1150) contains $427,200$ ALMs, which limits the DSPs usage to $1,369$. So our peak performance for any application on this board is $\frac{1369}{4} \times 2 \times 288.35 = 197$ Gflops/s for double precision. Considering the DSPs from Table 16, we know that fexchem peak performance is 7.4 Gflops/s.

We are using the experiments from (FAICT; D'HOLLANDER; GOOSSENS, 2019) regarding the bandwidth. According to Intel documentation, HARP2 bandwidth could achieve 30 GB/s if we considered the two PCIe channels and the QPI channel. A realistic benchmark showed that the bandwidth is 16 GB/s for different scenarios. Our peak performance is achievable with that bandwidth if we have an arithmetic intensity of 0.43. In other words, we need at least 5 flops for each double retrieved from memory. Our algorithm has 6 flops for 32 bytes, an arithmetic intensity of 0.1875 or 3 Gflops/s. The roofline model clearly shows that parallelism can improve performance slightly. Our profiling shows that the actual performance is 263 Mflops/s ($6 flops \times 227.08 Mhz \times 10^6 \times 0,1933$).

From now on, we must define two profiling properties relevant to the performance discussion: Occupancy and Bandwidth efficiency. **Occupancy** close to 100% means we can process a loop iteration every cycle. **Bandwidth efficiency** close to 100% means we are using all the data we retrieved from global memory. For fexchem, we have an occupancy of 19.33% with an efficiency of 100%, which means we are processing an iteration every 5 cycles and using all data fetched from global memory.

We also have to point out the compiler problems with those designs. For some reason, the compiler struggles to manage stop conditions among the loops. We are using the constant values from the kernel-imposed serial execution between the loops in 11 and 18 lines (Source

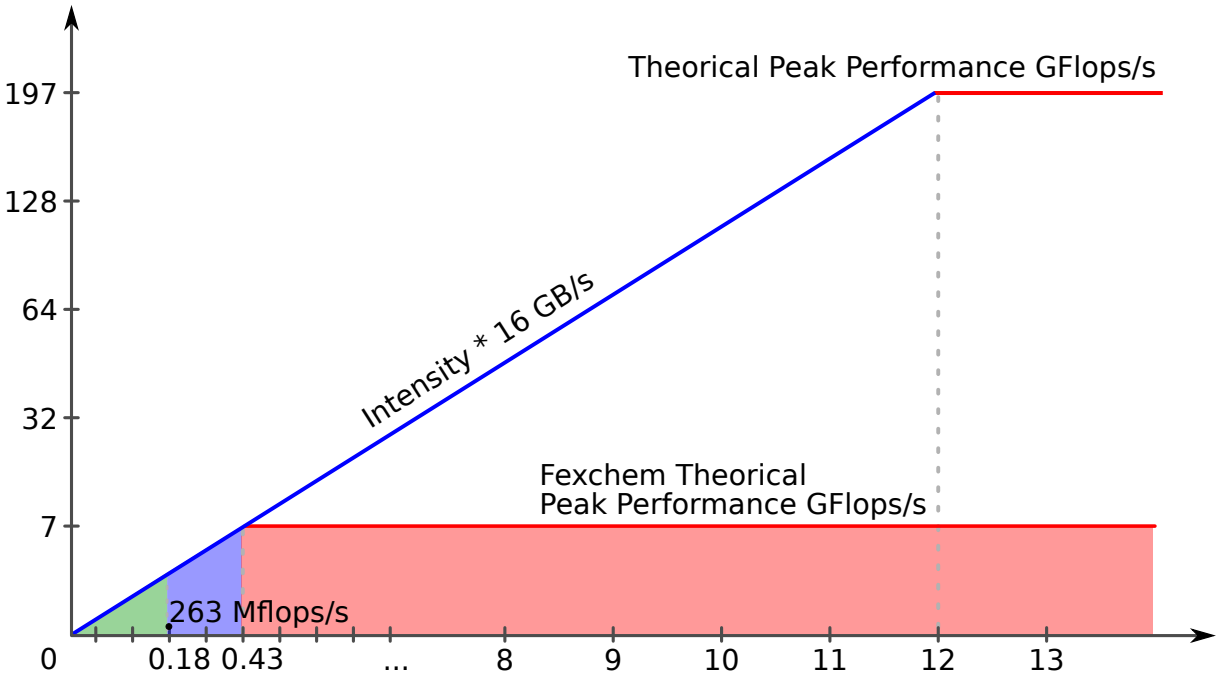

Figure 21 – Roofline model for the Fexchem Function.

Table 16 – Hardware resources for fexchem with automatic banking

	Automatic banking	**Custom banking**
Registers	192,857 (11%)	192,278 (11%)
Logic	122,322 (29%)	122,507 (29%)
DSPs	65 (4%)	66 (4%)
RAM blocks	954 (35%)	684 (25%)
Maximum Clock Frequency (MHz)	210	227.08

Code 2) because it found a dependency on them. We could solve this by using integer literals unsuitable for generic designs. This problem occurred several times in different kernels.

5.3.4 Dratedc

Generating A depends on two functions: (1) Jacdchemdc – Jacobian, and (2) Dratedc – a derivative of reaction rates. Here it is important to point out that the Jacobian matrix is unrelated to the Jacobian Method (iterative method for solving linear systems). Dratedc has a sparse matrix of doubles, whose size is $BLOCK\ SIZE \times REACTIONS \times GASES$. Considering RELACS TUV as the chemical mechanism, those values are $65 \times 128 \times 47$, respectively.

According to our previous results, the Jacobian matrix has 96% of sparsity, that is, 3 MB of unused memory, without considering the compiler's replication. Such a matrix has a fixed sparse structure, where all the used indices are known upfront the execution, which makes it suitable for flattening to 1D and removing the zero values. Furthermore, all the matrix definitions are in Fortran 90, and we needed C-like arrays, so we used PRI to extract the indices and generate a flattened version of those arrays with NoN-Zero elements (NNZ) only.

This solution requires two tables of indices with NNZ size for each and one for storing the results. Dratedc is represented in Equation 5.3. Y factor is not always needed, so we padded with 1 whenever it is missing. For RELACS TUV, that means 3.46 kB compared to almost 3 MB before our scripts. We also list it in Source Code 3.

$$dw(i) = rk(j) * y(k) \quad \forall i = 1, 2, \ldots, nnz. \tag{5.3}$$

Source code 3 – Dratedc – Derivative of Reaction Rates in OpenCL (custom banking)

```
1  void dratedc(const unsigned int block_end, const unsigned int nnz_dw,
2               const unsigned int nreactions, const unsigned int nspecies,
3               double local_rk[block_end][nreactions],
4               double local_y[block_end][nspecies + 1], double local_dw[nnz_dw]) {
5      const unsigned int rk_indices[222] = {...};
6      const unsigned int y_indices[222] = {...};
7      for (unsigned int block = 0; block < 65; block++) {
8        for (unsigned int y = 0; y < 222; y++) {
9            local_dw[y] = local_rk[block][rk_indices[y]] * local_y[block][y_indices[y]];
10       }
11     }
12 }
```

We also performed some experiments on automatic banking vs. custom banking. All the results are in Table 17. The performance results are nearly identical, with a few block RAMs saved. Again, the clock did not significantly impact the overall performance; we explain that with the roofline discussion. Improving throughput with loop unrolling did not work; the compiler could not generate a correct design for some reason, even though the compiler steps finished normally. We were able to find this bug while executing the design.

Table 17 – Hardware resources for dratedc

	Automatic banking	**Custom banking**
Registers	158,613 (9%)	159,664 (9%)
Logic	105,038 (25%)	105,561 (25%)
DSPs	5 (< 1%)	8 (< 1%)
RAM blocks	596 (22%)	568 (21%)
Maximum Clock Frequency (MHz)	225	262.5

We can explain the performance results with the roofline model. For that, we repeat the method applied in Section 5.3.3 and represent the model in figure 22. According to the roofline, the peak performance for dratedc is 1 Gflop/s, but that goes even lower when considering the arithmetic intensity, which is around 0.041 (or 666 Mflops/s). Our profiling shows that the efficiency is 100% and the actual performance is 107 Mflops/s due to the low occupancy (41.16%). Our results confirm that this algorithm does not allow any parallelism exploration due to its memory-bound nature. This study should be enough to support the statement that Rosenbrock should be entirely implemented on the FPGA side. However, we decided to implement it in the FPGA to avoid doubt.

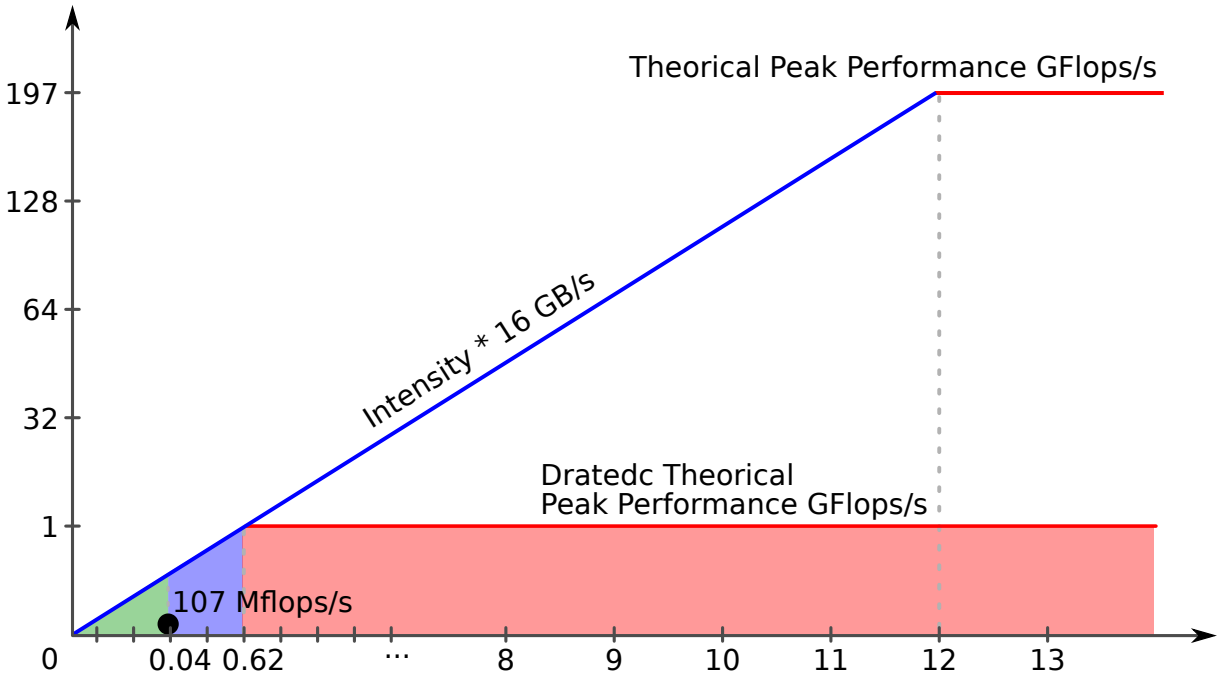

Figure 22 – Roofline model for Dratedc Function.

5.3.5 Jacobian

Our previous modified dratedc function has a new indexation due to the flattening and sparsity removal. To fix that problem, we used PRI again to remap all the indices to this new indexation, which required six matrices for the correct indexation and almost 20 kB of local memory to store constant data (we are also not considering the memory replication). Jacobian can be represented as in the Equation 5.4 and listed in the Source Code 4.

$$Jacc(i,j) = \sum_{x=1}^{N} dw(k) * const(index\_const(k)) \tag{5.4}$$

Where:

I = 1, 2, ..., species
J = 1, 2, ..., species
N = Number of derivative of reaction rate per $Jacc_{ij}$
K = Index extracted and remapped by the customized parser.

Source code 4 – Jacc – Jacobian in OpenCL

```
1 void jacc_fexchem_func(const int block_end, const int nreactions, const int nspecies,
2 const int nnz_dw, double local_rk[block_end][nreactions],
3 double local_y[block_end][nspecies + 1],
4 double local_jacc[block_end][nspecies][nspecies],
5 double local_dlr[block_end][nspecies]) {
6
7     double local_dw[222];
8     const unsigned int index_jacc[461][2] = {...};
```

```
    const int index_dw[1120] = {...};
    const double const_dw[564] = {...};
    const int index_const_dw[1120] = {...};
    const int index_dw_per_jacc[461] = {...};
    const int y_indices[222] = {...};

    for (unsigned int block_index = 0; block_index < block_end; block_index++) {
        for (unsigned int y = 0; y < nnz_dw; y++) {
            local_dw[y] = local_rk[block_index][rk_indices[y]] *
                          local_y[block_index][y_indices[y]];
        }

        unsigned int index_dw_begin = 0, index_dw_accum = 0,
                     index_w_begin = 0, index_w_accum = 0;
        for (unsigned int y = 0; y < NNZ_JACC; y++) {
            unsigned int dw_per_jacc = index_dw_per_jacc[y];
            double shift_reg_jacc[II_CYCLES_ACCUM + 1];

            #pragma unroll
            for (unsigned int i = 0; i < II_CYCLES_ACCUM + 1; i++) {
                shift_reg_jacc[i] = 0;
            }

            for (unsigned int z = 0; z < ITERATIONS; z++) {
                double accum_jacc = 0;

                #pragma unroll
                for (unsigned int i = 0; i < FACTOR; i++) {//data-parallelism
                    unsigned int index = z * FACTOR + i;
                    accum_jacc += ((index < dw_per_jacc) ?
                                   local_dw[index_dw[index_dw_begin]] *
                                   const_dw[index_const_dw[index_dw_begin]] : 0);
                    index_dw_begin++;
                }
                shift_reg_jacc[II_CYCLES_ACCUM] = shift_reg_jacc[0] + accum_jacc;

                #pragma unroll
                for (unsigned int i = 0; i < II_CYCLES_ACCUM; i++) {
                    shift_reg_jacc[i] = shift_reg_jacc[i + 1];
                }
            }
            double sum_jacc = 0;

            #pragma unroll
            for (unsigned int i = 0; i < II_CYCLES_ACCUM + 1; i++) {
                sum_jacc += shift_reg_jacc[i];
            }

            local_jacc[block_index][index_jacc[y][0]][index_jacc[y][1]] = -sum_jacc;

            index_dw_accum += dw_per_jacc;
            index_dw_begin = index_dw_accum;
        }
    }
}
```

Herein, we explore the same design to check whether the memory layout impacts the area and performance. We changed the input indices (*y_indices_fexchem* and *index_jacc*) to improve

memory alignment through PRI. Since Jacobian and Fexchem use a similar loop structure, we merged both algorithms (listed in Source Code 5) into the same loops and compared them to the Jacobian-only solution.

Source code 5 – Jacc + Fexchem (custom memory layout)

```
void jacc_fexchem_func(const int block_end, const int nreactions, const int nspecies,
    const int nnz_dw, double local_rk[block_end][nreactions],
    double local_y[block_end][nspecies + 1],
    double local_jacc[block_end][nspecies][nspecies],
    double local_dlr[block_end][nspecies]) {
    ...
    for (unsigned int block_index = 0; block_index < block_end; block_index++) {
        for (unsigned int y = 0; y < nnz_dw; y++) {
            local_dw[y] = local_rk[block_index][rk_indices[y]] *
                          local_y[block_index][y_indices[y]];
            if (y < nreactions){
                local_w[y] = local_rk[block_index][y] *
                             local_y[block_index][y_indices_fexchem[y][0]] *
                             local_y[block_index][y_indices_fexchem[y][1]];
            }
        }

        unsigned int index_dw_begin = 0, index_dw_accum = 0,
                     index_w_begin = 0, index_w_accum = 0;
        for (unsigned int y = 0; y < NNZ_JACC; y++) {
            unsigned int dw_per_jacc = index_dw_per_jacc[y];
            unsigned int w_per_chem = ((y < nspecies) ? index_w_per_chem[y] : 0);
            double shift_reg_jacc[II_CYCLES_ACCUM + 1];
            double shift_reg_chem[II_CYCLES_ACCUM + 1];

            #pragma unroll
            for (unsigned int i = 0; i < II_CYCLES_ACCUM + 1; i++) {
                shift_reg_jacc[i] = 0;
                shift_reg_chem[i] = 0;
            }

            for (unsigned int z = 0; z < 6; z++) {
                double accum_jacc = 0, accum_chem = 0;

                #pragma unroll
                for (unsigned int i = 0; i < FACTOR; i++) {//data-parallelism
                    unsigned int index = z * FACTOR + i;
                    accum_jacc += ((index < dw_per_jacc) ?
                                    local_dw[index_dw[index_dw_begin]] *
                                    const_dw[index_const_dw[index_dw_begin]] : 0);

                    accum_chem += ((index < w_per_chem) ?
                                    local_w[index_w[index_w_begin]] *
                                    const_w[index_const_w[index_w_begin]] : 0);
                    index_dw_begin++;
                    index_w_begin++;
                }
                shift_reg_jacc[II_CYCLES_ACCUM] = shift_reg_jacc[0] + accum_jacc;
                shift_reg_chem[II_CYCLES_ACCUM] = shift_reg_chem[0] + accum_chem;

                #pragma unroll
                for (unsigned int i = 0; i < II_CYCLES_ACCUM; i++) {
                    shift_reg_jacc[i] = shift_reg_jacc[i + 1];
```

```
53                      shift_reg_chem[i] = shift_reg_chem[i + 1];
54                  }
55              }
56              double sum_jacc = 0;
57              double sum_chem = 0;
58
59              #pragma unroll
60              for (unsigned int i = 0; i < II_CYCLES_ACCUM + 1; i++) {
61                  sum_jacc += shift_reg_jacc[i];
62                  sum_chem += shift_reg_chem[i];
63              }
64
65              local_jacc[block_index][index_jacc[y][0]][index_jacc[y][1]] = -sum_jacc;
66              if (y < nspecies){
67                  local_dlr[block_index][y & 0x2E] = sum_chem;
68              }
69
70              index_dw_accum += dw_per_jacc;
71              index_dw_begin = index_dw_accum;
72
73              index_w_accum += w_per_chem;
74              index_w_begin = index_w_accum;
75          }
76      }
77  }
```

Merging Jacobian and Fexchem imposes some extra computation for Jacobian, so we need to put some conditions before summation. The compiler manages this merged version much better than the Jacobi-only version. According to our results, the merged version is 13% faster than the Jacobi-only version. Managing the local memory layout also improved the results in 33%. We show the resource usage in Table 18, and the memory layout results in Table 19. Considering that, we proceeded with our experiments with the merged version.

Table 18 – Hardware resources for Jacobian only and merged Jacobian + Fexchem

	Jacobian only	**Merged**
Registers	193,164 (11%)	230,054 (13%)
Logic	122,900 (29%)	141,716 (33%)
DSPs	66 (4%)	126 (8%)
RAM blocks	1,764 (65%)	1,887 (70%)
Maximum Clock Frequency (MHz)	151.25	183.3

Table 19 – Hardware resources for Jacobian + Fexchem

	Automatic banking	**Custom banking**
Registers	245,615 (14%)	230,054 (13%)
Logic	150,912 (35%)	141,716 (33%)
DSPs	124 (8%)	126 (8%)
RAM blocks	1,844 (68%)	1,887 (70%)
Maximum Clock Frequency (MHz)	180	183.3

From the previous algorithm, we already know that dratedc is pushing performance down. That is also exposed in the roofline model presented in Figure 24. The jacobian only has a peak performance of 4.9 Gflops/s and arithmetic intensity of 0.208 or 3.3 Gflops/s. From those results, we can conclude that 3 more flops for the same 24 bytes would be enough to achieve the peak performance. Our profiling shows a sustained 101 Mflops/s performance due to its occupancy of 13.46%; note that the performance is similar to dratedc.

Regarding the merged version, we obtained conflicting results with reality. According to the analysis, it needs 56 bytes for 11 flops, an arithmetic intensity of 0.196 or 3.14 Gflops/s (performance lower than the Jacobian only). Our profiling shows that the actual performance is 225 Mflops/s for an occupancy of 11.2%, slightly higher than dratedc. As the results pointed out, the memory-bound functions do not benefit from the parallel logic in the FPGA and consume scarce resources. We still forced a Rosenbrock implementation to ensure the results were precise.

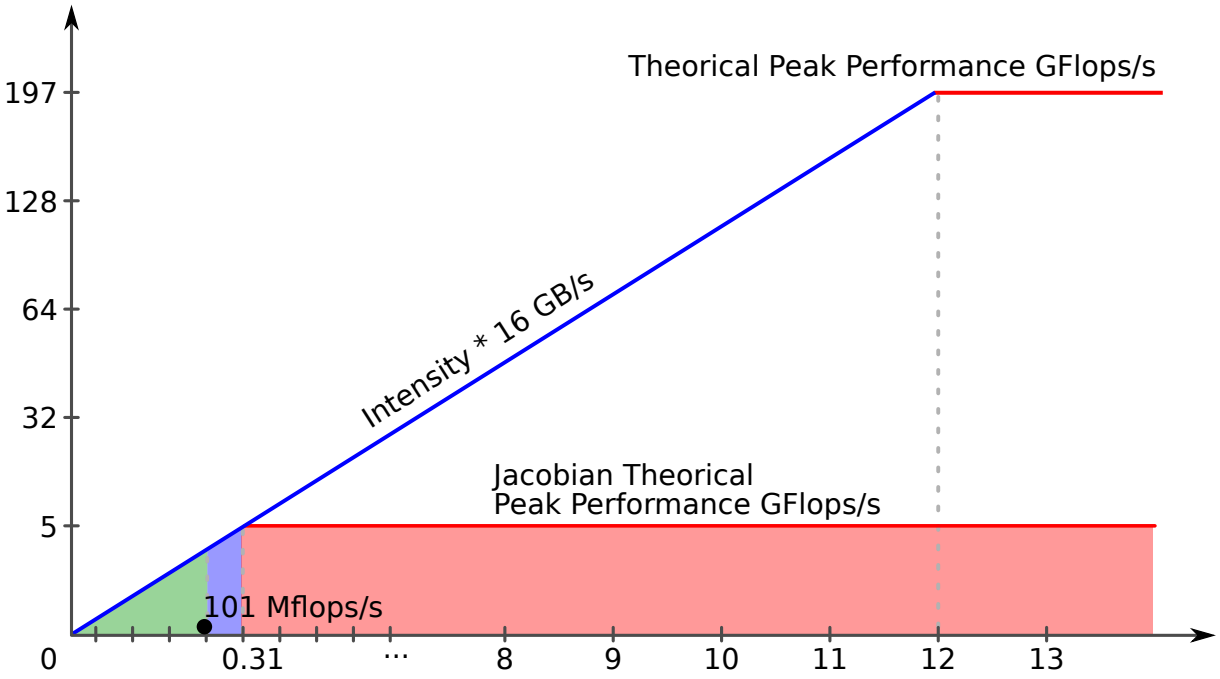

Figure 23 – Roofline model for Jacobian Function.

5.3.6 Rosenbrock

For implementing Rosenbrock, we have used two approaches: (1) Feeding the initial data to the FPGA and implementing a solution with memory-bound functions in the FPGA; (2) Avoiding memory-bound functions and implementing a streaming version of the Rosenbrock with the stages running in pipeline. In the following sections, we discuss the implementation of both.

The first approach avoids communication with the CPU at the cost of implementing memory-bound functions on the FPGA side. As we showed in the previous results, there are more efficient ways to solve this problem, leading us to the second solution, the streaming Rosenbrock.

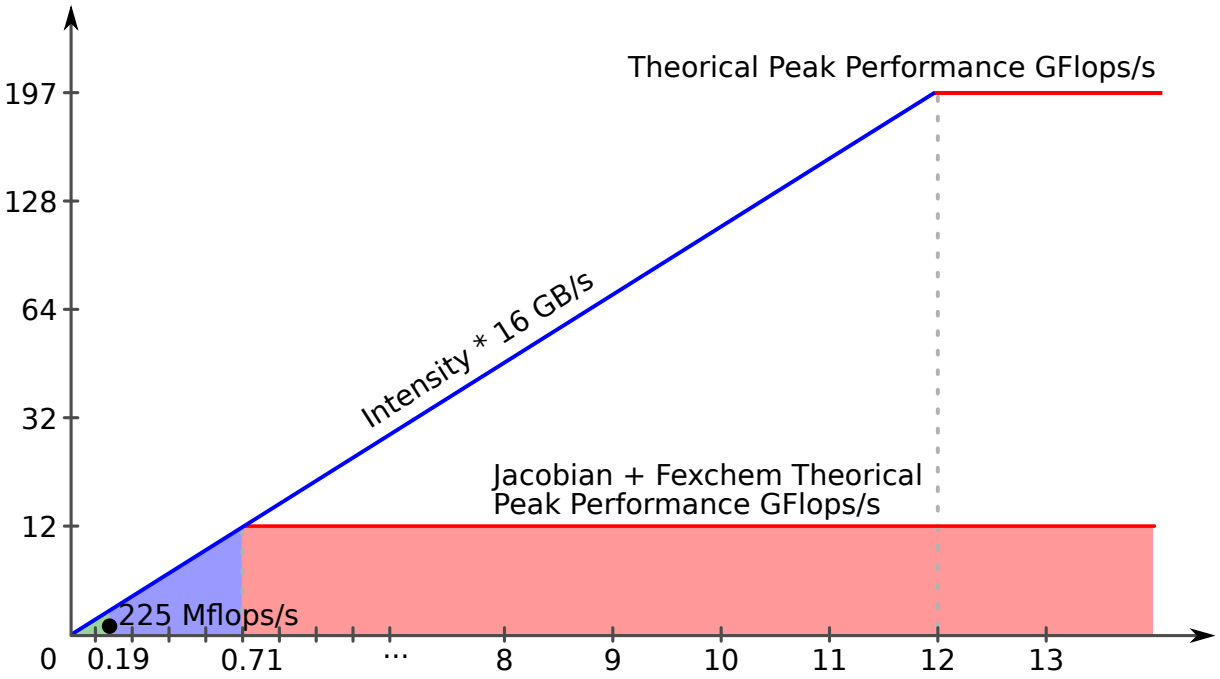

Figure 24 – Roofline model for Jacobian + Fexchem Functions.

In this second solution, we accepted the communication with the CPU as part of the solution and explored the parallelism across the Rosenbrock stages.

5.3.6.1 *Rosenbrock with memory-bound functions*

In this first version, we could not fit the entire Rosenbrock algorithm on the FPGA. Although we had all of them implemented, they did not fit on the board, which forced us to use only the first stage of the Rosenbrock. In this implementation, we compute the Jacobian and Fexchem matrices and use them as inputs to the QR decomposition design to solve the linear systems $Ax = b$. We still perform some analysis regarding the algorithm's behavior, design exploration of the memory layout, and the roofline model for the proposed solution as we did in all previous algorithms.

Our memory-bound function took so many resources that we had to constrain the QR decomposition for fitting the Arria 10 FPGA. For that, we disabled the loop unrolling of the dot product and replaced it with shift registers (see Source Code 6), which incurs less parallelism and possibly lower performance. We also had to improve memory utilization used by the upper triangular matrix R, which can work by using half of the initial memory at the cost of complex exit conditions.

Source code 6 – Dot product with shift register in OpenCL

```
1 double dot_product(unsigned int block_end, unsigned int nspecies, unsigned int k,
2                    unsigned int j, unsigned int block,
3                    double a[block_end][nspecies][nspecies]){
4     double shift_reg[II_CYCLES_ACCUM + 1];
5
```

```
6       #pragma unroll
7       for (unsigned int i = 0; i < II_CYCLES_ACCUM + 1; i++) {
8           shift_reg[i] = 0;
9       }
10
11      for (unsigned int m = 0; m < nspecies; m++) {
12          shift_reg[II_CYCLES_ACCUM] = shift_reg[0] + a[block][m][k] * a[block][m][j];
13
14          #pragma unroll
15          for (unsigned int i = 0; i < II_CYCLES_ACCUM; i++) {
16              shift_reg[i] = shift_reg[i + 1];
17          }
18      }
19
20      double sum = 0;
21      #pragma unroll
22      for (unsigned int i = 0; i < II_CYCLES_ACCUM + 1; i++) {
23          sum += shift_reg[i];
24      }
25
26      return sum;
27  }
```

Such changes were reflected in Rosenbrock's performance, as shown in the roofline model in Figure 25. Table 20 shows that the design exploration of the memory layout is not enough to provide much improvement either. In the roofline model, we observe that generating the jacobian and the chemical production on the hardware side have a peak performance of $3.71 GB/s$ with an arithmetic intensity of 0.23. As we can observe from the profiling results, our sustained performance is 16 Mflops/s with a poor occupancy of 0.42%.

Those results confirm that memory-bound functions should be on the software counterpart and not using resources of the FPGA. That is why we implemented a streaming Rosenbrock version, executing only the four stages of the Rosenbrock without the memory-bound functions. See Appendix B for more details.

Table 20 – Hardware resources for Rosenbrock

	Automatic banking	**Custom banking**
Registers	343,367 (20%)	337,884 (20%)
Logic	195,998 (35%)	193,958 (33%)
DSPs	215 (8%)	215 (8%)
RAM blocks	2411 (89%)	2408 (89%)
Maximum Clock Frequency (MHz)	145	154.7

5.4 Phase 4 – Streaming Rosenbrock

For the streaming Rosenbrock, we almost had to design the algorithm from scratch. Much of the previous results could not be used in this new solution. To the best of our knowledge, there is no such solution available for FPGAs in the literature, and it is one of the contributions of this

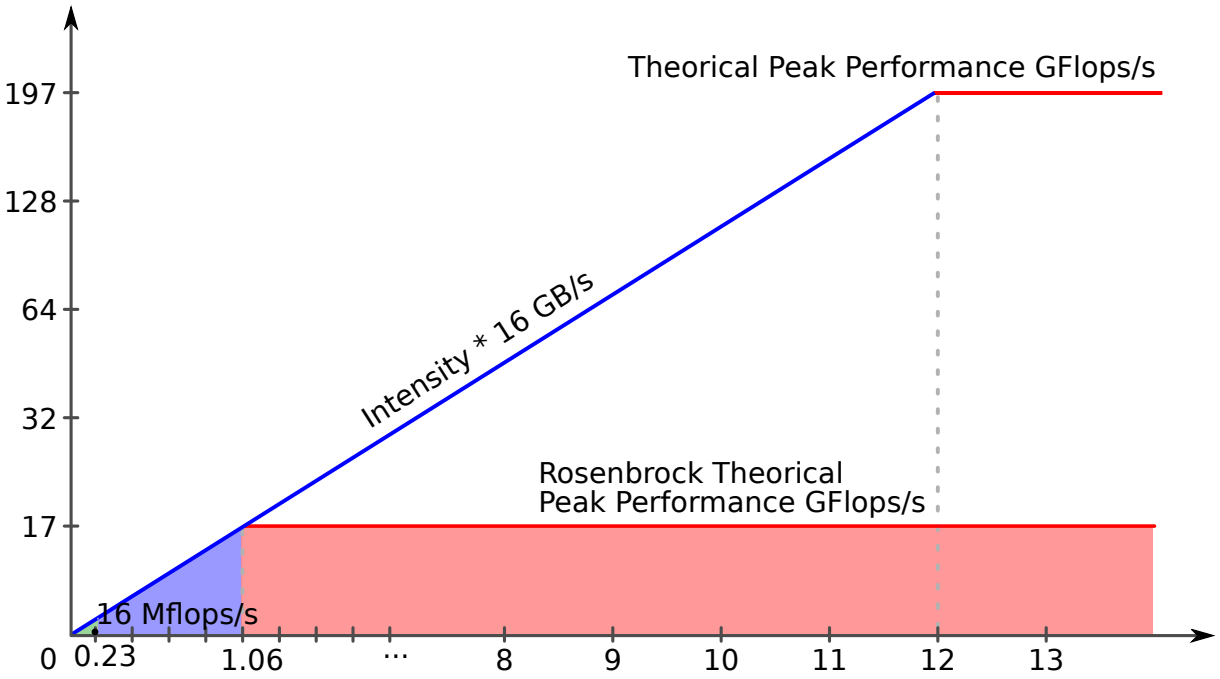

Figure 25 – Roofline model for Rosenbrock.

thesis. In this new version, we removed the Jacobian memory-bound algorithm. According to our studies of the bottlenecks, computing the four stages is the most time-demanding. We can increase the arithmetic intensity and parallelism by computing the four stages in a streaming fashion. Such streaming demands more than one matrix stored on the global memory.

It is found in the literature the parallelism across the stages. However, our Rosenbrock variation has dependencies over the stages. We overcame that by processing such stages in a pipeline fashion. Once the pipeline of stages is full, we have four stages executing in parallel and receiving data in a streaming fashion (represented in Figure 26). Our final circuit is represented in Figure 27.

Since we removed the Jacobian function, we had to communicate the initial conditions, which, as expected, is faster than generating it on the FPGA side. So first, we had to copy the content from global memory to local memory, which is necessary to avoid RAM contention and stalls over the pipeline. The first stage of Rosenbrock is the most expensive because we need to factorize the matrix.

Due to area restrictions, we had to modify the published algorithm. We implemented the same mechanism we used for the previous algorithms and let the programmer define whether they want data parallelism. According to our experiments, the board does not support unrolling factor over 3 for our solution due to the memory replication and exhaustion of the memory blocks (See the results for different unrolling factors in Appendix C). Since we are working with streaming and pipeline, that should only cause some delay for the first matrix because of the longer datapath.

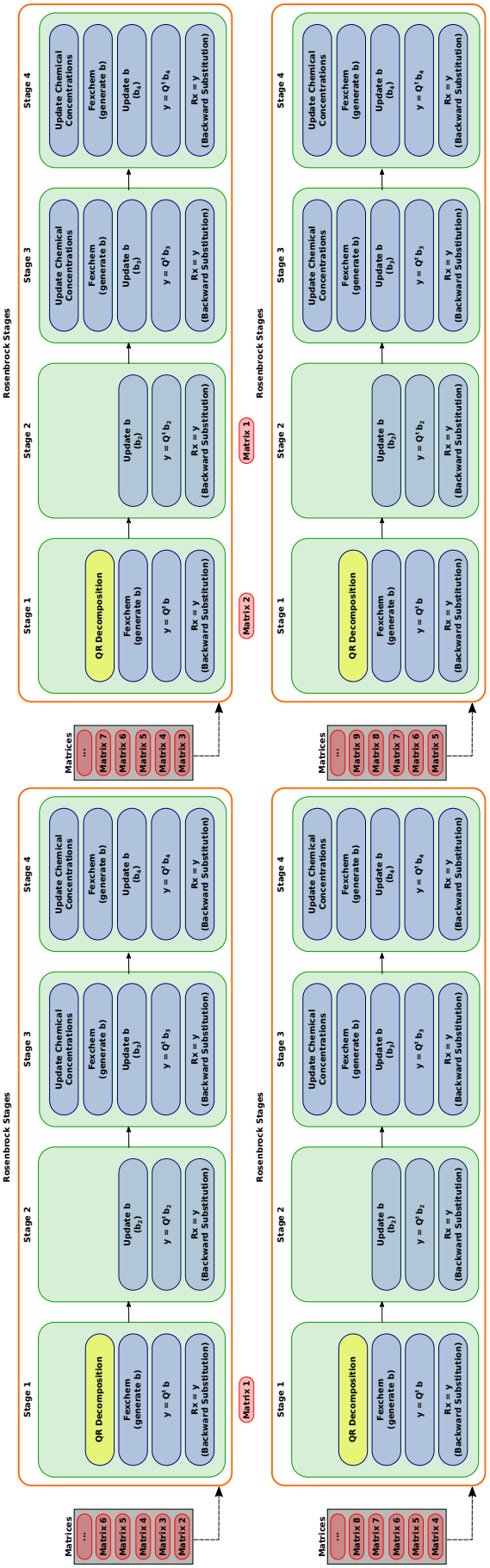

Figure 26 – New matrices are streamed at the same rate as the current stage of the Rosenbrock processes the previous ones. The QR factorization (yellow box) was implemented during the BEPE internship.

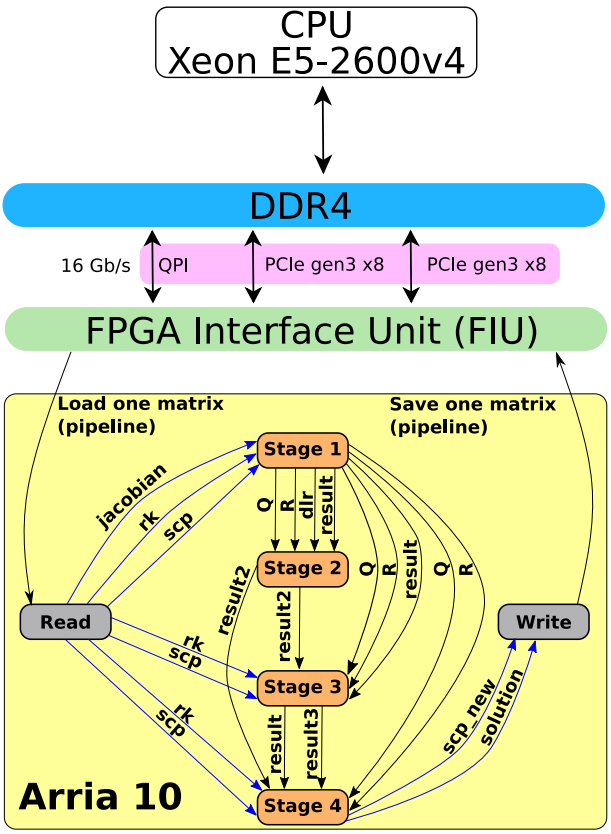

Figure 27 – Architecture for the streaming Rosenbrock. Read and Write vertices are the only ones communicating with the global memory. The remaining vertices and edges only communicate through non-blocking channels, that is, FIFOs implemented in the local memory.

After matrix decomposition, we have to compute fexchem, a memory-bound function. The Fexchem function is the same from Section 5.3.3. Once we have the decomposition from matrix A and generated array b with fexchem, we solve the linear system with QR and execute the backward substitution. Each stage executes the instructions in the following order: (I) Update chemical concentrations, (II) Generate b array, (III) Update b array with the previous result, (IV) Q transposed multiplied by b, (V) backward substitution.

The streaming Rosenbrock outputs the results and stores them on the global memory. We perform the error computation on the host side and check whether we need a new iteration. If a new iteration is needed, we communicate the updated initial conditions again, and the FPGA performs the computation as before until it converges to the expected error.

Herein, we show our initial results with this design. Our streaming version requires double the logic and registers from the FPGA. As we mentioned, decreasing data parallelism on the dot product is essential to improving memory usage and the slight difference in DSP resources. We show the results for hardware resources in Table 21.

Regarding performance, our initial results show almost 4× improvement over the published results. Those results are also competitive with the vectorized software version compiled with "-O3", which uses parallel vector instructions. Our design is still 5× slower than its software

Table 21 – Hardware resources for the Streaming Rosenbrock

	Streaming Rosenbrock
Registers	602,456 (35%)
Logic	291,998 (61%)
DSPs	278 (18%)
RAM blocks	2,284 (84%)
Maximum Clock Frequency (MHz)	158.93

counterpart. For the power consumption, we have used an estimation tool provided by Intel that approximates the consumed energy for the HARP 2. According to this tool, this design consumes around 24.6W.

We show the performance comparison between the vectorized software and the streaming design in Table 22. We did not show the communication time because we could hide the communication latency by overlapping computation with communication, which prevents us from timing the communication itself.

We also analyzed the roofline model for our streaming Rosenbrock in Figure 28. Table 23 shows each stage's flops and the bytes required for computation. For this analysis, we consider that each stage is independent and can write to the output its peak performance, which is 44.16 Gflops/s. Since our hardware has a peak performance of 22 Gflops/s, that is the maximum achieved by our circuit in theory. Our profiling shows a sustained performance of 11.2 Gflops/s ($75 flops \times 158.93 Mhz \times 10^6 \times 0,943$ occupancy) or 50% of the hardware capacity. Our occupancy is 94.3% with 100% of bandwidth efficiency, which means we are almost using the full potential of the bandwidth and processing an iteration every clock cycle.

Asynchronism among the stages is the reason for the performance loss. Although the FIFOs allowed us to hide most of the latency, they still have some stalls caused by the different performances for each stage. The reports also showed many replications for the on-chip memories, directly correlating to the degree of parallelism across the stages. The compiler replicated memory because that was the only way to keep the stages running in parallel.

Table 22 – Results for performance

	Software	**Streaming**
Send (μs)	-	-
Solve (μs)	4,329	23,588
Receive (μs)	-	-
Total (μs)	4,329	23,588

Nevertheless, on the Asynchronism, Table 23 shows the performance for each stage. Such differences demand more local memory as a buffer to avoid stalls among the stages, which exhausts the amount of on-chip memory. From these arithmetic intensity results, we noticed that stage 2 is the fastest because this kernel has a single communication with the global for fetching

Table 23 – Arithmetic intensity for each kernel

	Flops	Bytes	I	I x Bandwidth (Gflops/s)
Stage 1	27	48	0.56	9
Stage 2	10	8	1.25	20
Stage 3	16	32	0.5	8
Stage 4	22	48	0.45	7.3
Total	75	136	2.76	44.16

a double constant (8 bytes). OpenCL optimizes it to store on the on-chip memory as soon as it receives it. From these results, we noticed that Arria 10 does not allow further parallelism since we almost used the maximum number of RAM blocks. So we wanted to explore the same circuit in a more prominent architecture, so we chose Stratix 10.

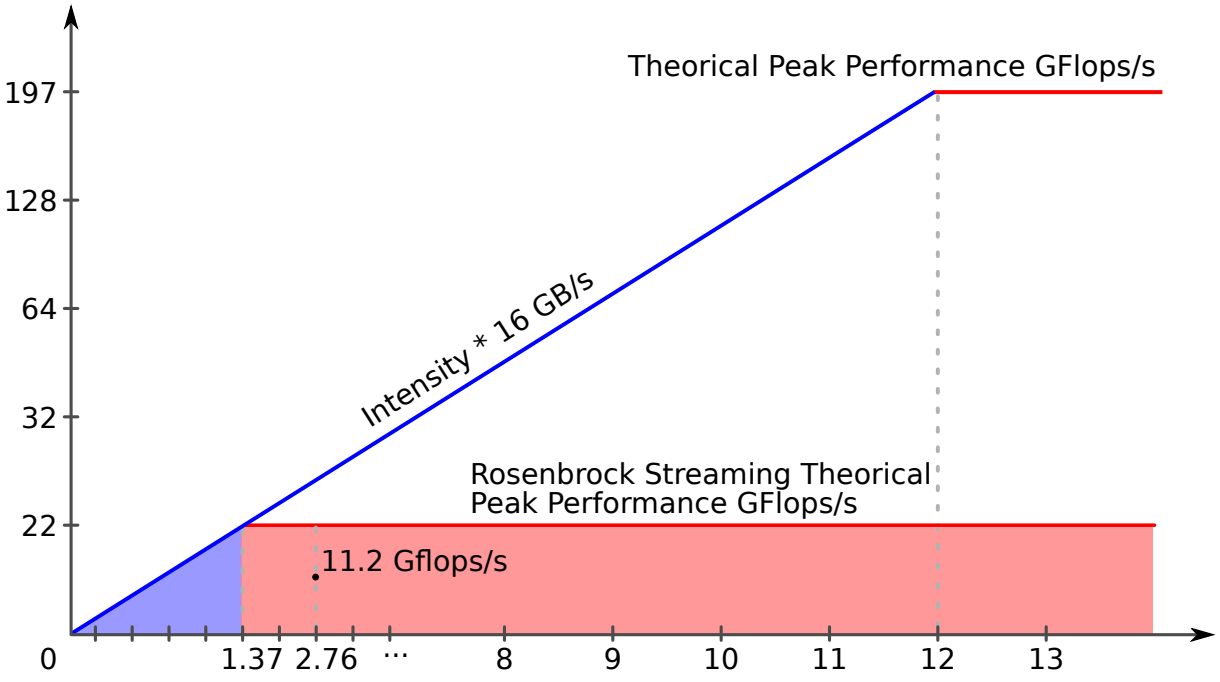

Figure 28 – Roofline model for the streaming Rosenbrock.

5.5 Phase 5 – Streaming Rosenbrock in the Stratix 10

Stratix 10 has more than twice the number of logic compared to Arria 10 on HARP 2 and 4× more RAM blocks. Stratix 10 also uses one of the latest compilers for Intel OpenCL. In this project, we use the 21.4 version (Arria 10 uses 16.0). We expected more performance than Arria since we use a faster architecture with an updated compiler. Besides, Stratix 10 gives us plenty of space to explore parallelism and memory replication.

As a first approach, we used the same source code developed for the Arria 10 FPGA. That did not work because the updated compiler complained about the data structure. From now

on, the programmer cannot use multidimensional arrays as parameters and must use a pointer to the multidimensional array. Some function signatures also changed, but the behavior is the same.

Although it compiled without problems, the design is not functional because of a deadlock between the read and write kernel on the global memory. That happened because Stratix 10 does not have an FPGA Interface Unit (FIU) as Arria 10, which allows the runtime chooses one of the three paths for fetching data from DDR4 global memory (see Figure 27). So, while a path is blocked for the reader, another is used for writing data, preventing the deadlock. Among the options to solve such a problem, we first tried to use clEnqueueMigrateMemObjects from the OpenCL API. This command guarantees that the data is sent to the device before its use, but that did not work either. For a second approach, we implemented a mutex that forces reading or writing to the global memory depicted in Figure 29.

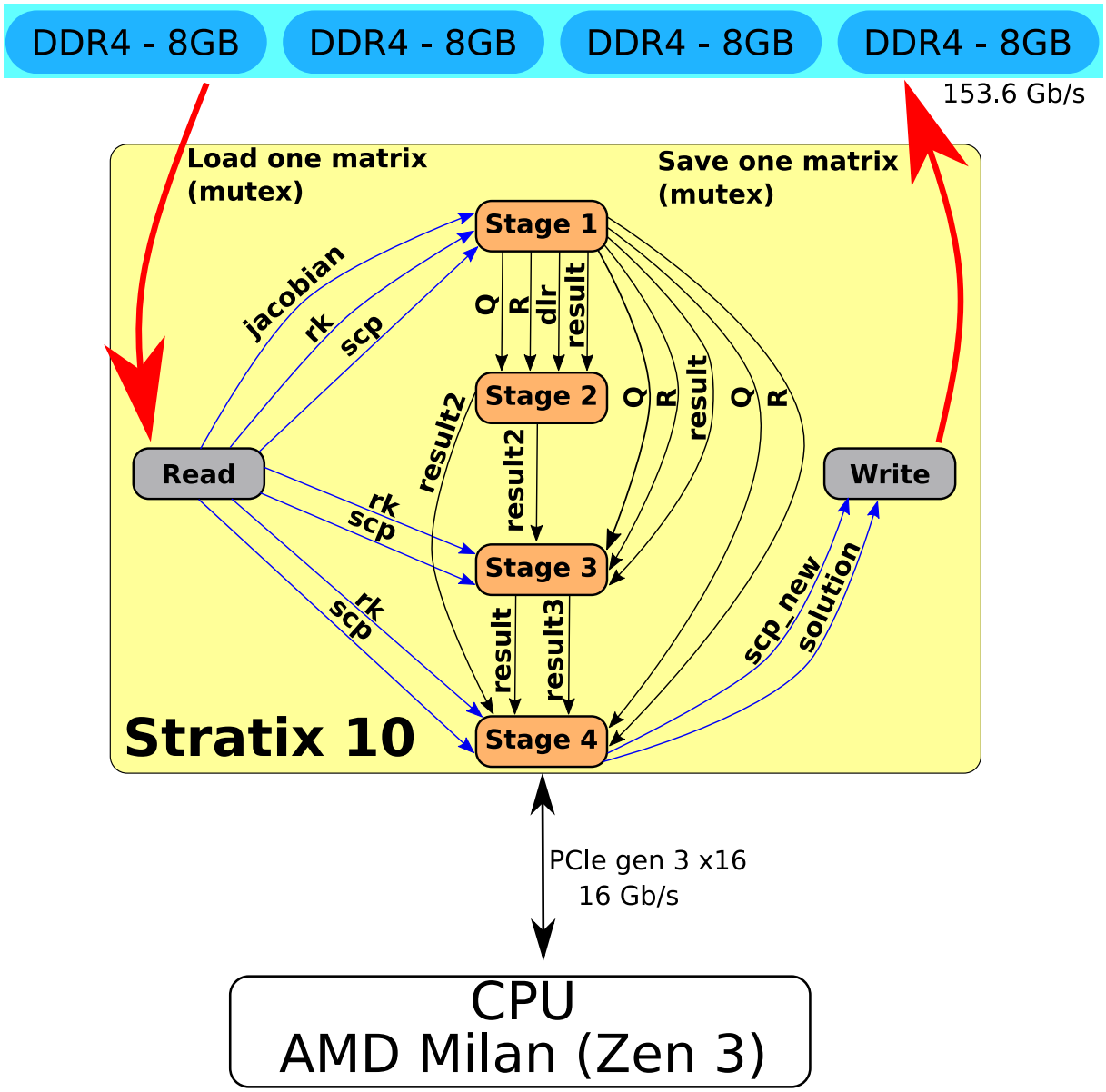

Figure 29 – Streaming Rosenbrock architecture for Stratix 10.

Using a mutex imposes a serial execution on the read and write kernels, which is acceptable as a first functional approach. However, we must improve it to provide a parallel solution like Arria's. For readability, we inserted the following results: (I) software result, (II) result for Arria 10, (III) result for mutex on Stratix 10, and (IV) result for local memory only on Stratix 10. We list the results for resource utilization in Table 24 and timing execution in Table 25.

The results indicate a severe performance drop (around 187×

That solution allowed us to read and write in parallel without deadlock at the cost of 8% more RAM blocks. The show results in Table 24 in the following order: (I) Streaming Rosenbrock on the Arria 10, (II) Streaming Rosenbrock with Mutex on the Stratix 10, (III) Streaming Rosenbrock with on-chip memory only on the Stratix 10. To our surprise, that solution saved logic usage and improved circuit frequency. As we can see, we improved execution in 2% even though the frequencies are much higher in the Stratix 10.

Table 24 – Resource usage for Arria 10 (I) and Stratix 10 (II) and (III)

	(I)	**(II)**	**(III)**
Registers	602,456 (35%)	882,968 (23%)	829,511 (22%)
Logic	291,998 (61%)	417,065 (45%)	396,200 (42%)
DSPs	278 (18%)	315 (5%)	320 (6%)
RAM blocks	2,278 (84%)	2,401 (20%)	3,328 (28%)
Frequency (Mhz)	159.93	260.0	267.5

Table 25 – Results for performance on the Stratix 10

	(I)	**(II)**	**(III)**	(IV)
Send (μs)	-	-	-	-
Computation (μs)	4,329	23,588	5,070,865	4,965,642
Receive (μs)	-	-	-	-
Total in HW (μs)	4,329	23,588	5,070,865	4,965,642

Those results did not make sense, so we decided to deep dive into OpenCL strategies for the Stratix 10 and found some problems in our design applied to this architecture. We will briefly discuss each of them in the following list:

1. **Reduce channel overhead**: we are already using non-blocking channels to avoid resource overhead;

2. **Reduce the number of kernels**: That is impossible since we need each stage in a separate kernel. Otherwise they will run serially;

3. **Using non-blocking channels**: loop control logic becomes more complex and might limit performance. That is also impossible to change without causing deadlock among the stages since they are asynchronous;

4. **Optimizing loop control**: Intel implemented a loop control specifically for Stratix 10, which decoupled the exit condition calculation from the loop's body. That comes at the price of a few extra cycles for flushing the loop whenever the loop exit condition is signaled. For designs with a few iterations, that becomes a bottleneck, for example, small matrices. BRAMS has matrices with 47×47; as we discussed previously, that is the maximum number we can process in production;

5. **Simplifying memory access to local memories**: Stratix 10 compiler discourages double pumping in the M20K memories, that is, M20K works as a 4-port memory running at twice the frequency. For that, the programmer must assign an attribute for each local memory, which does not guarantee any performance since that requires some try-and-error strategy. We did not change this attribute since we have $4\times$ more RAM blocks.

Intel provides more strategies in their documents, but we restrict them to the ones we consider essential to our design. As we see from this list, our design is inappropriate for the Stratix 10 architecture. We rely on separate kernels and 22 channels for communication among the stages, which all require a complex exit condition. Even Intel optimization for hyperflex loop control is decreasing performance. That means we would have to design a new design optimized for Stratix 10 from scratch, which we will leave for future work.

Although we did not improve performance, we measured the power consumption of this solution on Stratix 10 and compared it to the Arria 10. It is important to remind that Arria 10 provided an experimental tool that outputs an approximation of the power consumption for HARP 2. Since this is the only way to compare, we will use those results.

Table 26 – Energy consumption for computing Rosenbrock for 65 matrices of 47×47

	Time (μs)	**Power (W)**	**TDP (W)**	**Energy (J)**	**Energy TDP (J)**
Rosenbrock Arria 10	23,588	24.6	70	0.58	1.65
Rosenbrock Stratix 10	4,965,642	70	225	347.59	1117.26
Rosenbrock in Software	4,329	28	290	0.121	1.25

The results show that Arria 10 on HARP 2 is $611\times$ more efficient than the Stratix 10 FPGA on the Padderborn server. Regarding the software, we are still $5\times$ less energy efficient and around $5\times$ slower. First, we can justify those values by using a dense representation for QR Decomposition instead of a sparse representation; modifying to a sparse representation requires deeper optimizations because of the irregular access among the loops. From our experience, OpenCL for Intel FPGAs does require pipeline-friendly modifications. So, changing the data structure is not viable if the designer wants to improve performance. OpenCL compiler infers serial execution whenever the loop is irregular, considerably dropping performance.

5.6 Final Remarks

In this chapter, we discuss the results obtained in this chapter. Our results were split into 5 phases, where the first one is derived from our master's thesis. For the development, we have used three versions of OpenCL: (1) 16.1 for Stratix V, (2) 16.0 for HARP2, and (3) 21.4 for Stratix 10. Using a single version for the three architectures is impossible because that depends on the BSP provided by the board manufacturer. Except for Phase 1, all implementations used task parallelism in OpenCL because that is the closest language paradigm that leverages the FPGA resources.

Regarding the FPGA architectures, they were chosen because they are the top FPGAs in the market. Even Stratix V, an FPGA from 2010, is still competitive, although much smaller than the other two we used for this work. It is important to remark that HARP2 is available only for researchers around the globe, and Intel has already been warning about updating those Broadwell architectures. That is one of the reasons that their power tool for Arria 10 is an approximation; they do not intend to support this product anymore. Therefore, we moved our design to a newer architecture with more resources.

As we mentioned in the results, the source code required a few changes because Arria 10 from HARP2 and Stratix 10 from Paderborn are entirely different architectures. Even the FPGA architecture is different with its hyperflex registers. Exploring performance over different devices with the same source code is challenging, and it was not possible to maintain the performance with OpenCL portable language. The language portability depends on the manufacturer compiler and how they implement specific warnings according to the underlying architecture.

Another experience with those compilers showed us that loop unrolling is only sometimes efficient, especially if the design considers the algorithm's behavior. Most of the time, unrolling drops the performance because the compiler needs to duplicate the memory so the computation over the data can run in parallel. In Arria 10, the compiler uses double pumping to avoid replication by duplicating the clock frequency. Stratix 10 has this feature disabled by default because it contains $4\times$ more M20K memory than Arria 10, and Intel does not want to drop the performance by decreasing the frequency. Stratix 10 can reach up to 1 Ghz, so doubling the frequency is not an option.

Generally, our FPGA results are less energy efficient and perform less than the software counterpart. Stratix 10 had the worst performance related to our solution, which needs to be optimized for this architecture. Our results are still not a viable option for substituting the sequential software based on Sparse 1.3a, and we still need to further test with other chemical mechanisms with more reactions and, consequently, more data for processing. We must also note that the software version implements an optimized algorithm in a sparse format. At the same time, our solution is based on a dense representation so that we could use a pipeline for the algorithm's loops.

Fortran 90 source code used for converting is used in production for weather forecasting in Brazil. That is not a benchmark designed specifically for exploring the performance over a specific architecture, which provides a challenging application that needs newer solutions based on GPUs. This thesis provides an initial design exploration for solving ODEs raised from the chemical reactivity problem.

Regarding energy, our Arria 10 final architecture has the most promising results. We generated a streaming Rosenbrock method with 24.6W of power, which is almost $3\times$ less power than Stratix 10. Considering the efficiency, our results point to $426\times$ less joule per operation. Although we did not improve the results compared to the software version, we still argue that we use the dense representation of the matrices. This kind of data representation incurs in $10\times$ more data for processing. Extrapolating the current results for Arria 10 for $10\times$ fewer data, we would have 0.058 joules per operation, $2\times$ less energy per operation than the software counterpart. On this extrapolation, we consider that the sparse implementation would have the same performance as the dense one.

CHAPTER

6

CONCLUSION

Besides the author's previous research, that explored the use of the Jacobian iterative method, we also have "A hardware/software co-design framework for the dynamics module of the Brazilian weather forecast model - BRAMS" (PEREIRA, 2019). The work presented here is one of the first studies that focuses on porting BRAMS to a heterogeneous computing architecture in a hardware/software codesign.

We continued Pereira's effort at porting BRAMS modules to a heterogeneous architecture comprising FPGAs and CPUs. The research started with performing profiling analysis and exploring BRAMS' source code. That was a long activity that took years from us because of the structure and the complex equations. BRAMS is a complex application with over 400.000 lines of source code that uses advanced fluid mechanics, physics, chemistry, and parallel processes that communicate with each during the execution. The output of BRAMS is a set of products generated through the configuration file. We relied on an expert's advice on setting this configuration file since the number of options can quickly become unmanageable.

During this extended analysis, we concluded that chemical reactivity is one of the hot spots that could leverage parallel solutions (Chapter 4). Although the module has plenty of concurrency, it also imposes a challenge due to the stiff ODE equations requiring implicit methods for solving efficiently. Our studies also showed that modeling such applications is complex and requires a deep knowledge of meteorological physics and chemistry. So we decided to develop hardware/software codesign from scratch to solve the stiff ODEs raised by CCATT-BRAMS. Our efforts were towards the Rosenbrock Method, the current solver implemented in BRAMS.

According to Sartori (2014), the Rosenbrock method was designed to allow vector operations so the programmers can extract concurrency when porting their applications to a new underlying architecture. The method is also designed to guarantee numerical stability and avoid tiny timesteps. That was one of the reasons that we wanted to extend this implementation to

leverage the FPGA resources.

This thesis presented a hardware/software codesign targeting the ODE solver for the chemical reactivity problem in BRAMS. We provided this solution for Intel's heterogeneous architecture in HARP 2. Before implementing the codesign solution, we needed to extract the Rosenbrock method and its linear solver algorithms. Considering that BRAMS is a complex application that depends on several libraries and a huge amount of data, we had to refactor the source code of BRAMS to a smaller, manageable source code. This first refactoring resulted in 5.000 lines of source in Fortran 90 and C, a fraction of the entire CCATT-BRAMS application. That is one of the reasons that very few works concentrate their efforts on porting legacy applications to heterogeneous architectures. Most of those applications depend on legacy languages and high human effort for building the hardware/software codesign solution for those environments.

We could define our hardware/software codesign from this source code extraction. Our final solution can be split into two parts: (1) matrix decomposition for the linear solver, (2) Hardware/Software codesign for the Rosenbrock Method. Those solutions went through 5 phases so we could find a balance between hardware and software. Those implementations took a long time because we had to adapt several source codes to OpenCL. We also had to rely on a parser for generating structure in C-like to avoid error-prone activities such as copying the fexchem constants to a file. Implementing in OpenCL is much faster than implementing in RTL.

We performed initial studies with VHDL, and the difficulties with floating-point operations and timing constraints made us change our focus (see Appendix A). Implementing an implicit algorithm in VHDL demands much more work and human effort than using a High-Level Synthesis language. Although the final results with the streaming Rosenbrock solution did not improve the serial execution, we still provided a solution targeting FPGAs. It is important to remind that the software solution uses sparse representation, and our solution uses dense representation. Considering that only 10% of the matrix is non-zero, the software is computing $10\times$ fewer data than our architecture solution.

Moving to a higher-density logic FPGA did not improve our results; it became even worse because we needed to implement a specific solution for the new architecture. Those 5 phases exposed the heavy memory-bound operations over the Rosenbrock implicit method. That could be better explored with future FPGA technology like Agilex, which can provide up to 102 GB/s of peak memory performance against our 16 GB/s on Arria 10 in HARP2. The new hyperflex architecture also promises to work well on non-optimized OpenCL kernels, which can be promising for future work with our streaming solution.

Overall, OpenCL is competitive in generating efficient hardware. We developed a solution that is $5\times$ slower, but we must consider that we are processing the zero elements. So we are processing $10\times$ more data than the software solution. Our final architecture consumes 4 W less than the CPU execution. Considering the extrapolation of those results, our final architecture performs with 0.058 of power efficiency compared to the 0.121 of the software, which is almost

2× more power efficient. We also need to consider that we are using an old OpenCL compiler version based on 16.0, which is impossible to change because of the BSP development. We also assessed our solution with real data from a huge application in Brazil. The development of this research proved that FPGAs are mature enough to be used on high-performance applications and not only restricted to embedded systems.

6.1 Contributions

The main contribution of this thesis is implementing a hardware/software codesign for the ODE implicit solver based on the Rosenbrock Method. Thanks to the parser application that generates the structures in C-like, we also provide an optimal balance between hardware and software. This approach allowed us to move to the FPGA and perform a good trade-off between memory-bound and CPU-bound operations. Other contributions:

- Providing a solution that was converted from a Fortran 90 legacy application to a codesign of hardware/software;
- Providing a QR decomposition solution for FPGAs in OpenCL, which can be used for any application that can fit on the target FPGA and is not restricted to the sizes of the BRAMS matrices. This work was published in "Exploration of FPGA-Based Hardware Designs for QR Decomposition for Solving Stiff ODE Numerical Methods Using the HARP Hybrid Architecture";
- To the best of our knowledge, this is the first porting of the Rosenbrock method to Intel's heterogeneous architecture;
- To the best of our knowledge, this is the first work to use the Stratix 10 hyperflex architecture and expose the new challenges for the Rosenbrock method;
- Provided OpenCL kernels that can be adapted for other FPGA architectures;
- Provided a parser that allows the programmer to choose the chemical mechanism. This parser also allows some local memory savings, which is the most used resource for this implementation;
- Provided a good trade-off between memory-bound and CPU-bound operations which is key for performance.

6.2 Limitations

- There is still room for improvement over the QR decomposition regarding the matrix size;

- The work did not explore a sparse solution because that would break the loop pipeline. So we could explore more solutions that could perform a trade-off between pipeline parallelism and less data-intensive operations;
- We did not explore more implicit ODE due to the modeling complexity of the application;
- The system was tested for a single process. We did not implement collective calls for computing the matrices from the other core processors. That is a normal limitation on external computing devices. It is the programmer's responsibility to solve that;
- We were not able to couple our solution to BRAMS because it requires administrative permissions and because there are no FPGAs available at CPTEC/INPE;
- Using the FPGAs remotely did incur more time for implementation;
- Compiling the green part (the area responsible for the BSP) and the blue part (the user logic) is impossible. Intel locked this feature in HARP 2;
- Old OpenCL compiler version.

6.3 Lessons Learned

While initially planed, some aspects of this work were no explored. These included exploring different floating-point types, implementing a VHDL solution, and using OpenCL only for communication with the CPU. We also had yet to learn how much work involves in modeling an implicit solver for CCATT-BRAMS. The most challenging work was understanding the BRAMS application due to the lack of documentation, and the ones found are accessible to people that are not experts in the field. Working with remote FPGAs also imposed some extra time whenever there was a bug or maintenance on the server. We had to contact the INPE personnel and travel to their institute to better understand how the application works.

The remote execution of those tests was not planned either. We were considering coupling our solution to BRAMS as we did in the master thesis, but then we had to face the limitations of a shared server and extract the source code from BRAMS. This task was error-prone and time-consuming, which took much more time than planned, mainly because we needed to fully understand the application. We also needed to adapt the source to execute the same operations on BRAMS with the same performance.

6.4 Future Work

Several kinds of research could be derived from this thesis. The first and most important should be related to the efficient sparse implementation of the QR decomposition, a data-intensive

operation that needs a pipeline solution. Such a solution could change the results obtained by this work.

Exploring advanced implicit methods targetting FPGAs is another direct study from this thesis. The literature review showed that we still have plenty of algorithms to explore, most of which are implemented targetting GPUs. Modeling a new solution considering CCATT-BRAMS and FPGAs is also interesting, but it requires a specialist in the field working with a hardware specialist. Another work we propose is to explore Agilex architecture, and this is an FPGA targetting data-intense operations with peak memory performance 6× higher than our best solution with Intel's heterogeneous architecture. Table 27 shows a resource comparison among the FPGA architecture we worked on.

Table 27 – Resource comparison

	Arria 10	**Stratix 10**	**Agilex**
Logic elements	1,150,00	2,753,00	3,851,520
DSPs	1,518	5,760	9,375
Local Memory	65	244	311

We should also consider using Intel OneAPI to replace the current OpenCL SDK, the most updated language for High-Level Synthesis. Our previous studies on this language showed that it is possible to migrate to OneAPI from OpenCL; although the languages are not identical, they have similar data structures. Even Intel has a tutorial page for migration. In this author's opinion, it is weird that Intel accepted a language that merges host and kernel into a single file. Anyone who has already worked with OpenCL High-Level Synthesis knows that compiling the kernel is time-demanding and should be avoided. However, that is impossible when there is a single file for the host and kernel. Any change on the host side will incur in hours of compilation. Thankfully, that is not the only mode allowed by OneAPI.

Another feature that requires special attention is the higher-level abstraction for data transfers. Our previous results with OpenCL show that this is not recommended for hardware design if the final target is performance. Companies may argue that it is a programmer-friendly environment, so the adoption of FPGA can increase. However, why would a programmer adopt something slower and harder to optimize? The best results we obtained in this thesis required deep knowledge of hardware, and most of them were possible by avoiding the good practices standard in the software environments.

One huge limitation is that OneAPI for Intel FPGAs can only be used with Intel devices. That may impose some adoption challenges, especially for legacy applications like BRAMS. For the last, we found it curious that Intel does not support volatile type in OneAPI kernels, which was critical for performance in HARP 2 architecture since CCI-P still presents some bugs. Initial research showed that OneAPI is a viable solution for stencil computation and not optimized for tiny matrices (RODRIGUEZ-CANAL *et al.*, 2021), which is precisely the type of matrix inside the CCATT-BRAMS module.

BIBLIOGRAPHY

ACKERMAN, P.; ACKERMAN, S.; KNOX, J. **Meteorology**. Jones & Bartlett Learning, LLC, 2013. ISBN 9781284027389. Available: <https://books.google.com.br/books?id=qWcrAQAAQBAJ>. Citation on page 37.

ALIKHANIA, J.; MASSOUDIEHB, A.; BHOWMIKA, U. Gpu-accelerated solution of activated sludge model's system of odes with a high degree of stiffness. In: . [S.l.: s.n.], 2017. p. 555–560. Citation on page 63.

ALTERA. **Implementing FPGA Design with the OpenCL Standard**. [S.l.], 2013. Available: <https://www.altera.com/en_US/pdfs/literature/wp/wp-01173-opencl.pdf>. Accessed: May 4, 2017. Citations on pages 13, 53, and 54.

AMOS, D.; LESEA, A.; RICHTER, R. **FPGA-based prototyping methodology manual: Best practices in design-for-prototyping**. [S.l.]: Happy About, 2011. Citations on pages 13 and 41.

ANDERSON, E.; BAI, Z.; DONGARRA, J. Generalized qr factorization and its applications. **Linear Algebra and its Applications**, Elsevier, v. 162, p. 243–271, 1992. Citation on page 79.

ANDERSON, M. J.; SHEFFIELD, D.; KEUTZER, K. A predictive model for solving small linear algebra problems in gpu registers. In: IEEE. **2012 IEEE 26th International Parallel and Distributed Processing Symposium**. [S.l.], 2012. p. 2–13. Citation on page 61.

BAHI, J.; CHARR, J.-C.; COUTURIER, R.; LAIYMANI, D. A parallel algorithm to solve large stiff ode systems on grid systems. In: . [S.l.: s.n.], 2007. p. 534–541. Citation on page 63.

____. A parallel algorithm to solve large stiff ode systems on grid systems. **International Journal of High Performance Computing Applications**, v. 23, n. 2, p. 140–151, 2009. Citation on page 63.

BELARDINI, P.; BERTOLI, C.; CORSARO, S.; D'AMBRA, P. The impact of different stiff ode solvers in parallel simulation of diesel combustion. **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**, v. 3726 LNCS, p. 958–968, 2005. Citation on page 63.

BINDEL, D.; GOODMAN, J. Principles of scientific computing linear algebra ii, algorithms. 2006. Citation on page 79.

BITTWARE, I. **S5-PCIe-HQ**. [S.l.], 2015. 57 p. Citation on page 42.

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. Citation on page 57.

BLUESPEC. **BSV Documentation**. 2017. Available: <http://wiki.bluespec.com/Home/BSV-Documentation>. Accessed: Nov. 28, 2018. Citations on pages 13 and 49.

BOBDA, C. **Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications**. Springer Netherlands, 2007. ISBN 9781402061004. Available: <https://books.google.com.br/books?id=_cNSgjR32LkC>. Citation on page 41.

BOUT, D. V. **FPGAs?! Now What?** [s.n.], 2011. Available: <http://www.xess.com/static/media/appnotes/FpgasNowWhatBook.pdf>. Citation on page 41.

BRUDER, J. Numerical results for a parallel linearly-implicit runge-kutta method. **Computing (Vienna/New York)**, v. 59, n. 2, p. 139–151, 1997. Citation on page 63.

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. Citation on page 50.

CENTER, P. **Intel® Performance Counter Monitor - A Better Way to Measure CPU Utilization**. 2023. Available: <https://www.intel.com/content/www/us/en/developer/articles/technical/performance-counter-monitor.html>. Accessed: May 05, 2023. Citation on page 74.

CHAPRA, S. **Numerical Methods for Engineers**. [s.n.], 2014. ISBN 9780077492168. Available: <https://books.google.com.br/books?id=3GpzCgAAQBAJ>. Citations on pages 28 and 34.

CHASNOV, J. R. **Introduction to Differential Equations**. 2016. Available: <https://www.math.ust.hk/~machas/differential-equations.pdf>. Accessed: Dec 8, 2018. Citation on page 33.

CHIU, G. Using intel quartus prime software to maximize performance in the intel hyperflex fpga architecture. Intel white paper, 2021. Citation on page 44.

CHU, P. **FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version**. Wiley, 2011. ISBN 9781118210604. Available: <https://books.google.com.br/books?id=nXdbDRUUCyUC>. Citation on page 41.

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. Citation on page 37.

CURTISS, C. F.; HIRSCHFELDER, J. O. Integration of stiff equations. **Proceedings of the National Academy of Sciences**, National Acad Sciences, v. 38, n. 3, p. 235–243, 1952. Citation on page 28.

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. Citation on page 54.

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. Citations on pages 13 and 55.

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. Citation on page 60.

D'AMORE, R.; CIRCUITOS, V.-D. e síntese de. Digitais. **Editora LTC**, 2005. Citation on page 48.

DAVE, N. H. *et al.* **Designing a processor in Bluespec**. Phd Thesis (PhD Thesis) — Massachusetts Institute of Technology, 2005. Citation on page 49.

DAVISON, M. **Shallow/Deep Convection**. [S.l.], 1999. Available: <http://origin.wpc.ncep.noaa.gov/international/training/deep/sld001.htm>. Accessed: May 8, 2017. Citation on page 37.

DENNARD, R. H.; GAENSSLEN, F. H.; YU, H.-N.; RIDEOUT, V. L.; BASSOUS, E.; LEBLANC, A. R. Design of ion-implanted mosfet's with very small physical dimensions. **IEEE Journal of solid-state circuits**, IEEE, v. 9, n. 5, p. 256–268, 1974. Citation on page 29.

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. Citation on page 57.

FAICT, T.; D'HOLLANDER, E. H.; GOOSSENS, B. Mapping a guided image filter on the harp reconfigurable architecture using opencl. **Algorithms**, v. 12, n. 8, 2019. ISSN 1999-4893. Available: <https://www.mdpi.com/1999-4893/12/8/149>. Citations on pages 13, 42, 43, and 86.

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. Master's Thesis (Master's Thesis) — Instituto Nacional de Pesquisas Espaciais, 2014. Master's thesis at INPE-SP. Citations on pages 38 and 75.

FREITAS, S.; LONGO, K.; DIAS, M. S.; CHATFIELD, R.; DIAS, P. S.; ARTAXO, P.; ANDREAE, 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. Citation on page 36.

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. Citation on page 36.

GALLERY, R. Hardware/software codesign. **The ITB Journal**, v. 4, n. 1, p. 5, 2015. Citation on page 56.

GE, X.; ZHU, H.; YANG, F.; WANG, L.; ZENG, X. Parallel sparse lu decomposition using fpga with an efficient cache architecture. In: IEEE. **2017 IEEE 12th International Conference on ASIC (ASICON)**. [S.l.], 2017. p. 259–262. Citation on page 60.

GOLUB, G.; LOAN, C. V. **Matrix Computations**. Johns Hopkins University Press, 2013. (Johns Hopkins Studies in the Mathematical Sciences). ISBN 9781421407944. Available: <https://books.google.com.br/books?id=X5YfsuCWpxMC>. Citation on page 79.

GUPTA, P. **Xeon+FPGA Platform for the Data Center**. [S.l.], 2015. Available: <https://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf>. Accessed: Apr 10, 2017. Citation on page 56.

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. Citation on page 57.

GáCITA, M. S. **Estudos Numéricos de Química Atmosférica para a região do Caribe e América Central com Ênfase em Cuba**. Master's Thesis (Master's Thesis) — Instituto Nacional de Pesquisas Espaciais - INPE, São José dos Campos - SP - Brasil, 2011. Citation on page 38.

HAIRER, E.; NØRSETT, S.; WANNER, G. **Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems**. Springer, 1993. (Lecture Notes in Economic and Mathematical Systems). ISBN 9783540604525. Available: <https://books.google.com.br/books?id=m7c8nNLPwaIC>. Citation on page 28.

HOWARD, P. **Modeling with ODE**. 2009. Available: <www.math.tamu.edu/~phoward/m647/modode.pdf>. Citation on page 36.

HUTTON, M. Understanding how the new intel hyperflex fpga architecture enables next-generation high-performance systems. Intel white paper, 2022. Citations on pages 13 and 44.

IEEE. Verilog register transfer level synthesis. **IEC 62142-2005 First edition 2005-06 IEEE Std 1364.1**, p. 1–116, 2005. Citation on page 48.

INPE/CPTEC. **Model Description**. [S.l.], 2022. Available: <http://brams.cptec.inpe.br/about/>. Accessed: Dec. 4, 2022. Citation on page 36.

INTEL. **Intel FPGA SDK for OpenCL – Best Practices Guide**. [S.l.], 2016. Available: <https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf>. Accessed: Mar 29, 2017. Citations on pages 13 and 54.

____. **Intel FPGA SDK for OpenCL – Programming Guide**. [S.l.], 2016. Available: <https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf>. Accessed: May 4, 2017. Citation on page 54.

____. Fpga agility and flexibility for the data-centric world. Intel white paper, 2017. Citation on page 45.

____. **Floating-Point IP Cores User Guide**. [S.l.], 2021. Available: <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_altfp_mfug.pdf>. Accessed: Oct. 15, 2021. Citation on page 86.

____. **OpenCL™ Code Interoperability**. 2022. Available: <https://www.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top/software-development-process/composability/opencl-code-interoperability.html>. Accessed: Dec 11, 2022. Citation on page 47.

____. **Current and Recent FPGA Systems at PC2**. 2023. Available: <https://pc2.uni-paderborn.de/hpc-services/available-systems/fpga-research-clusters>. Accessed: May 05, 2023. Citation on page 74.

____. **PowerPlay Power Analyzer Support Resources**. 2023. Available: <https://www.intel.com/content/www/us/en/support/programmable/support-resources/power/sof-qts-power.html>. Accessed: May 05, 2023. Citation on page 77.

JACOB, D. **Introduction to atmospheric chemistry**. [S.l.]: Princeton University Press, 1999. Citation on page 33.

JACOB, D. J. **Chemical Tracer Models: an introduction**. 2007. Available: <http://acmg.seas.harvard.edu/education/jacob_lectures_ctms_chap1.pdf>. Accessed: Nov. 28, 2018. Citation on page 33.

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. Citations on pages 55 and 56.

JIANG, Z.; RAZIEI, S. A. An efficient fpga-based direct linear solver. In: IEEE. **2017 IEEE National Aerospace and Electronics Conference (NAECON)**. [S.l.], 2017. p. 159–166. Citation on page 61.

JUNIOR, C. Alberto Oliveira de S.; BISPO, J.; CARDOSO, J. M.; DINIZ, P. C.; MARQUES, E. Exploration of fpga-based hardware designs for qr decomposition for solving stiff ode numerical methods using the harp hybrid architecture. **Electronics**, MDPI, v. 9, n. 5, p. 843, 2020. Citations on pages 61, 73, and 130.

JUNIOR, M. B. d. S. **Portabilidade com Eficiência da Advecção do Modelo BRAMS entre Arquiteturas Multi-Core e Many-Core**. 92 p. Master's Thesis (Master's Thesis) — Instituto Nacional de Pesquisas Espaciais, 2015. Master's thesis at INPE-SP. Citation on page 30.

KALINNIK, N.; RAUBER, T. Execution behavior analysis of parallel schemes for implicit solution methods for odes. In: IEEE. **2018 17th International Symposium on Parallel and Distributed Computing (ISPDC)**. [S.l.], 2018. p. 1–8. Citation on page 63.

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. Citation on page 59.

KARP, M.; PODOBAS, A.; JANSSON, N.; KENTER, T.; PLESSL, C.; SCHLATTER, P.; MARKIDIS, S. High-performance spectral element methods on field-programmable gate arrays: implementation, evaluation, and future projection. In: IEEE. **2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)**. [S.l.], 2021. p. 1077–1086. Citation on page 45.

KHALAF, B.; AL-NEMA, M. Generalized parallel algorithms for bvps in odes. In: . [S.l.: s.n.], 2009. p. 759–765. Citation on page 63.

KREYSZIG, E. **Advanced Engineering Mathematics**. John Wiley & Sons, 2010. ISBN 9780470458365. Available: <https://books.google.com.br/books?id=UnN8DpXI74EC>. Citation on page 27.

KROSHKO, A.; SPITERI, R. Efficient simd solution of multiple systems of stiff ivps. **Journal of Computational Science**, v. 4, n. 5, p. 377–385, 2013. Citation on page 63.

KUNDERT, K. S.; SANGIOVANNI-VINCENTELLI, A. **Sparse1.3**. [S.l.], 1988. Available: <http://web.cs.ucla.edu/classes/CS258G/sis-1.3/sis/linsolv/>. Accessed: Oct. 28, 2015. Citations on pages 39 and 79.

LAI, Y.-H.; USTUN, E.; XIANG, S.; FANG, Z.; RONG, H.; ZHANG, Z. Programming and synthesis for software-defined fpga acceleration: status and future prospects. **ACM Transactions on Reconfigurable Technology and Systems (TRETS)**, ACM New York, NY, v. 14, n. 4, p. 1–39, 2021. Citation on page 47.

LAMBERS, J. **Lecture 9 Notes**. 2010. Lecture Note. Available: <http://www.math.usm.edu/lambers/mat461/spr10/lecture9.pdf>. Citation on page 34.

LANGHAMMER, M.; PASCA, B. High-performance qr decomposition for fpgas. In: **Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**. [S.l.: s.n.], 2018. p. 183–188. Citation on page 60.

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. Citation on page 75.

LINFORD, J. C.; SANDU, A. Vector stream processing for effective application of heterogeneous parallelism. In: **Proceedings of the 2009 ACM Symposium on Applied Computing**. ACM, 2009. (SAC '09), p. 976–980. ISBN 978-1-60558-166-8. Available: <http://doi.acm.org/10.1145/1529282.1529496>. Citation on page 28.

LIU, L.; WANG, H.; LIU, X.; JIN, X.; HE, W. B.; WANG, Q. B.; CHEN, Y. Greencloud: a new architecture for green data center. In: **Proceedings of the 6th international conference industry session on Autonomic computing and communications industry session**. [S.l.: s.n.], 2009. p. 29–38. Citation on page 29.

LONGO, K.; FREITAS, S.; PIRRE, M.; MARÉCAL, V.; RODRIGUES, L.; PANETTA, J.; ALONSO, M.; ROSÁRIO, N.; MOREIRA, D.; GÁCITA, M. *et al.* The chemistry catt–brams model (ccatt–brams 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. Citations on pages 13, 36, 37, 38, 75, and 76.

LUAN, V.; OSTERMANN, A. Parallel exponential rosenbrock methods. **Computers and Mathematics with Applications**, v. 71, n. 5, p. 1137–1150, 2016. Cited By 3. Citation on page 63.

MACINTOSH, H. J.; BANKS, J. E.; KELSON, N. A. Implementing and evaluating an heterogeneous, scalable, tridiagonal linear system solver with opencl to target fpgas, gpus, and cpus. **International Journal of Reconfigurable Computing**, Hindawi, v. 2019, 2019. Citation on page 61.

MARTINEZ, L. A. **Projeto de um sistema embarcado de predição de colisão e pedestres baseado em computação reconfigurável**. Phd Thesis (PhD Thesis) — Universidade de São Paulo, 2017. Citation on page 49.

MARTINEZ, L. A. **Um framework para coprojeto de hardware e software de sistemas avançados de assistência ao motorista baseados em câmeras**. Phd Thesis (PhD Thesis) — Universidade de São Paulo, 2017. Citation on page 57.

MARTINEZ, L. A. **Um framework para coprojeto de hardware e software de sistemas avançados de assistência ao motorista baseados em câmeras**. Phd Thesis (PhD Thesis) — University of São Paulo, 2017. Citation on page 49.

MATHWORKS. **Solve Stiff ODEs**. 2018. Available: <https://www.mathworks.com/help/matlab/math/solve-stiff-odes.html>. Citations on pages 13 and 35.

MENG, H.; WAKABAYASHI, K.; KURODA, T. A scalable linear equation solver fpga using high-level synthesis. In: . [S.l.: s.n.], 2022. (24th Workshop on Synthesis And System Integration of Mixed Information technologies). Citation on page 61.

MOORE ANDREW; WILSON, R. **FPGAs for Dummies**. Wiley, 2017. ISBN 978-1-119-39047-3. Available: <https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/misc/fpgas_for_dummies_ebook.pdf>. Citation on page 41.

MOORE, G. E. *et al.* **Cramming more components onto integrated circuits**. [S.l.]: McGraw-Hill New York, 1965. Citation on page 29.

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 ccatt-brams atmospheric chemistry model (jules-ccatt-brams1. 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. Citation on page 40.

MUNSHI, A. **The OpenCL Specification**. [S.l.], 2009. Available: <https://www.khronos.org/registry/OpenCL/specs/opencl-1.0.pdf>. Accessed: Mar 21, 2017. Citations on pages 13, 52, and 53.

MUNSHI, A.; GASTER, B.; MATTSON, T. G.; GINSBURG, D. **OpenCL programming guide**. [S.l.]: Pearson Education, 2011. Citations on pages 50 and 53.

MURALIDHAR, R.; BOROVICA-GAJIC, R.; BUYYA, R. Energy efficient computing systems: Architectures, abstractions and modeling to techniques and standards. **ACM Computing Surveys (CSUR)**, ACM New York, NY, v. 54, n. 11s, p. 1–37, 2022. Citations on pages 13 and 50.

MUSLIM, F. B.; MA, L.; ROOZMEH, M.; LAVAGNO, L. Efficient fpga implementation of opencl high-performance computing applications via high-level synthesis. **IEEE Access**, IEEE, v. 5, p. 2747–2762, 2017. Citations on pages 29 and 47.

NANE, R.; SIMA, V.-M.; PILATO, C.; CHOI, J.; FORT, B.; CANIS, A.; CHEN, Y. T.; HSIAO, H.; BROWN, S.; FERRANDI, F. *et al.* A survey and evaluation of fpga high-level synthesis tools. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 35, n. 10, p. 1591–1604, 2015. Citation on page 47.

NIEMEYER, K. E.; SUNG, C.-J. Accelerating moderately stiff chemical kinetics in reactive-flow simulations using gpus. **Journal of Computational Physics**, Elsevier, v. 256, p. 854–871, 2014. Citations on pages 29 and 62.

OFENBECK, G.; STEINMANN, R.; CAPARROS, V.; SPAMPINATO, D. G.; PÜSCHEL, M. Applying the roofline model. In: IEEE. **2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. [S.l.], 2014. p. 76–85. Citation on page 86.

PARKER, M.; MAUER, V.; PRITSKER, D. Qr decomposition using fpgas. In: IEEE. **2016 IEEE National Aerospace and Electronics Conference (NAECON) and Ohio Innovation Summit (OIS)**. [S.l.], 2016. p. 416–421. Citations on pages 60, 61, 72, and 80.

PEDRONI, V. A. **Circuit design with VHDL**. [S.l.]: MIT press, 2004. Citation on page 48.

PENG, R. **Algorithm design using spectral graph theory**. Phd Thesis (PhD Thesis) — Microsoft Research, 2013. Citation on page 79.

PEREIRA, E. d. S. **Um framework para coprojeto de hardware/software para o módulo da dinâmica do modelo brasileiro de previsão do tempo-BRAMS**. Phd Thesis (PhD Thesis) — Universidade de São Paulo, 2019. Citations on pages 57 and 107.

PETCU, D. Numerical solution of odes with distributed maple. **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**, v. 1988, p. 666–674, 2001. Citation on page 63.

RAHMANI, A. M.; LILJEBERG, P.; HEMANI, A.; JANTSCH, A.; TENHUNEN, H. The dark side of silicon. **Springer International Publishing, AG Switzerland**, Springer, v. 10, p. 978–3, 2016. Citation on page 29.

RANDALL, D. A. **An Introduction to Atmospheric Modeling**. [s.n.], 2013. Available: <http://kiwi.atmos.colostate.edu/group/dave/at604.html>. Citation on page 37.

RAYMOND, E. S. **The Cathedral & the Bazaar: Musings on linux and open source by an accidental revolutionary**. [S.l.]: " O'Reilly Media, Inc.", 2001. Citation on page 36.

RODRIGUEZ-CANAL, G.; TORRES, Y.; ANDÚJAR, F. J.; GONZALEZ-ESCRIBANO, A. Efficient heterogeneous programming with fpgas using the controller model. **The Journal of Supercomputing**, Springer, v. 77, n. 12, p. 13995–14010, 2021. Citations on pages 48 and 111.

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. Citation on page 60.

RUIZ, J. M.; LOPERA, J. O.; CARRILLO, J. Exploiting the multilevel parallelism and the problem structure in the numerical solution of stiff odes. In: IEEE. **Parallel, Distributed and Network-based Processing, 2002. Proceedings. 10th Euromicro Workshop on**. [S.l.], 2002. p. 173–180. Citation on page 63.

SANAULLAH, A.; HERBORDT, M. C. Fpga hpc using opencl: Case study in 3d fft. In: **Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies**. [S.l.: s.n.], 2018. p. 1–6. Citation on page 29.

SANDU, A.; VERWER, J.; BLOM, J.; SPEE, E.; CARMICHAEL, G.; POTRA, F. Benchmarking stiff ode solvers for atmospheric chemistry problems ii: Rosenbrock solvers. **Atmospheric environment**, Elsevier, v. 31, n. 20, p. 3459–3472, 1997. Citation on page 40.

SARTORI, L. M. **Métodos para resolução de EDOs stiff resultantes de modelos químicos atmosféricos**. Master's Thesis (Master's Thesis) — Universidade de São Paulo, 2014. Citations on pages 28, 40, 72, and 107.

ŠÁTEK, V. Stiff systems analysis. Citeseer, 2011. Citation on page 34.

SCHAUMONT, P. **A practical introduction to hardware/software codesign**. [S.l.]: Springer Science & Business Media, 2012. Citations on pages 13 and 56.

SEIFOORI, Z.; EBRAHIMI, Z.; KHALEGHI, B.; ASADI, H. Introduction to emerging sram-based fpga architectures in dark silicon era. In: **Advances in Computers**. [S.l.]: Elsevier, 2018. v. 110, p. 259–294. Citation on page 47.

SELICK, P. **Differential Equations I**. 2011. Available: <http://www.math.toronto.edu/selick/B44.pdf>. Accessed: Dec 8, 2018. Citation on page 33.

SHI, Y.; GREEN, W. H.; WONG, H.-W.; OLUWOLE, O. O. Accelerating multi-dimensional combustion simulations using gpu and hybrid explicit/implicit ode integration. **Combustion and Flame**, Elsevier, v. 159, n. 7, p. 2388–2397, 2012. Citations on pages 75 and 76.

SILVA, E. Pereira da. **Projeto de um Processador Open Source em Bluespec Baseado no Processador Soft-core Nios II da Altera**. 95 p. Master's Thesis (Mestrado em Ciência da Computação) — Univerisity of São Paulo, São Paulo, 2014. Citation on page 42.

SIMON, C. P.; BLUME, L. **Mathematics for economists**. [S.l.]: Norton New York, 1994. Citation on page 35.

SINGH, C. K.; PRASAD, S. H.; BALSARA, P. T. Vlsi architecture for matrix inversion using modified gram-schmidt based qr decomposition. In: IEEE. **20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07)**. [S.l.], 2007. p. 836–841. Citation on page 80.

SOUZA, C. A. O. D. **A hardware/software codesign for the chemical reactivity of BRAMS**. 112 p. Master's Thesis (Master's Thesis) — University of Sao Paulo, 2017. Master's thesis at ICMC-USP. Citation on page 76.

SOUZA, C. A. O. D.; PEREIRA, E. D. S.; MARQUES, E. A hardware/software codesign for the chemical reactivity of brams. In: IEEE. **2017 Euromicro Conference on Digital System Design (DSD)**. [S.l.], 2017. p. 70–77. Citation on page 59.

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. Citation on page 37.

STONE, C.; ALFERMAN, A.; NIEMEYER, K. Accelerating finite-rate chemical kinetics with coprocessors: Comparing vectorization methods on gpus, mics, and cpus. **Computer Physics Communications**, v. 226, p. 18–29, 2018. Citation on page 63.

STONE, C.; DAVIS, R. Techniques for solving stiff chemical kinetics on gpus. In: **51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition**. [S.l.: s.n.], 2013. p. 369. Citations on pages 28 and 75.

SUN, Y.; LIU, H.; ZHOU, T. Sparse cholesky factorization on fpga using parameterized model. **Mathematical Problems in Engineering**, Hindawi, v. 2017, 2017. Citation on page 61.

SUTHERLAND, S.; DAVIDMANN, S.; FLAKE, P. **SystemVerilog for Design Second Edition: A Guide to Using SystemVerilog for Hardware Design and Modeling**. [S.l.]: Springer Science & Business Media, 2006. Citation on page 49.

TABAK, J. **Mathematics and the Laws of Nature: Developing the Language of Science**. Facts on File, 2004. (Facts on File math library). ISBN 9780816049578. Available: <https://books.google.com.br/books?id=h5xguAAACAAJ>. Citations on pages 27 and 33.

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. Citation on page 56.

THOMA, Y.; DASSATTI, A.; MOLLA, D.; PETRAGLIO, E. Fpga-gpu communicating through pcie. **Microprocessors and microsystems**, Elsevier, v. 39, n. 7, p. 565–575, 2015. Citation on page 29.

TSOI, K. H.; LUK, W. Axel: A heterogeneous cluster with fpgas and gpus. In: **Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays**. [S.l.: s.n.], 2010. p. 115–124. Citation on page 29.

TSUCHIYAMA, R.; NAKAMURA, T.; IIZUKA, T.; ASAHARA, A.; SON, J.; MIKI, S. **The OpenCL Programming Book**. Fixstars, 2012. Available: <https://books.google.com.br/books?id=O86m1hJxA6QC>. Citations on pages 50 and 53.

VAUGHAN, C. **Deep Thoughts on Deep Convection**. [S.l.], 2009. Available: <http://blogs.ei.columbia.edu/2009/03/01/deep-thoughts-on-deep-convection/>. Accessed: May 8, 2017. Citation on page 37.

VELAGAPUDI, S. Addressing memory-bandwidth and compute-intensive challenges with intel agilex m-series fpgas. Intel white paper, 2022. Citation on page 45.

VERWER, J. G.; SPEE, E. J.; BLOM, J. G.; HUNDSDORFER, 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. Citation on page 38.

WANNER, G.; HAIRER, E. Solving ordinary differential equations ii. **Stiff and Differential-Algebraic Problems**, 1991. Citation on page 38.

____. **Solving ordinary differential equations II**. [S.l.]: Springer Berlin Heidelberg, 1996. Citation on page 75.

WON, M. S. Intel agilex fpgas deliver a game-changing combination of flexibility and agility for the data-centric world. Intel white paper, 2022. Citations on pages 13, 45, and 46.

WU, W.; SHAN, Y.; CHEN, X.; WANG, Y.; YANG, H. Fpga accelerated parallel sparse matrix factorization for circuit simulations. In: SPRINGER. **Intl. Symp. on Applied Reconfigurable Computing**. [S.l.], 2011. p. 302–315. Citation on page 60.

XILINX. **Xilinx Acquires AutoESL to Enable Designer Productivity and Innovation With FPGAs and Extensible Processing Platform**. 2011. Available: <https://www.prnewswire.com/news-releases/xilinx-acquires-autoesl-to-enable-designer-productivity-and-innovation-with-fpgas-and-extensible-processing-pla html>. Accessed: Dec 11, 2022. Citation on page 47.

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. Citation on page 63.

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. Citation on page 37.

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: <http://www.mdpi.com/2073-4433/2/3/510>. Citations on pages 28 and 75.

ZHUO, L.; PRASANNA, V. K. 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. Citation on page 60.

ZOHOURI, H. R.; PODOBAS, A.; MATSUOKA, S. Combined spatial and temporal blocking for high-performance stencil computation on fpgas using opencl. In: **Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**. [S.l.: s.n.], 2018. p. 153–162. Citation on page 75.

APPENDIX

A

EXPLICIT METHOD FOR THE PREDITOR-PREY PROBLEM

A.1 VHDL implementation

This set of experiments were done in 2019. In our master's degree research, we realized that OpenCL did not improve performance. We concluded the following reasons caused that:

1. Jacobi iterative method was not suitable for our problem;
2. Stratix V is not suitable for floating-point operations with double precision;
3. Processing a single matrix at a time;
4. OpenCL was not generating an efficient design.

All of those reasons were enough to convince us to explore a mixed approach, where we design a circuit with VHDL and uses OpenCL as a channel of communication with our host. Intel OpenCL has a feature where the programmer can define its circuit and insert it into the compilation process.

We decided to start with a simple example to test if the implementation would be feasible or not according to our schedule. For that, we chose a small stiff equation with two variables (for comparison, BRAMS model for air quality uses 47 variables), the preditor-prey problem. We represent our problem in the set of Equations A.1, which represents the number of rabbits and foxes over the time. For the experiments, we used the following initial values: $a = 10$, $b = -1$, $l = -0.1$, $k = 1$, R_0, and $F_0 = 5$.

$$\begin{cases} \left(\frac{dr}{dt}\right)_{rabbit} = aR - bRF & \text{(A.1a)} \\ \left(\frac{df}{dt}\right)_{fox} = -lF + kRF, & \text{(A.1b)} \end{cases}$$

Given the simplicity of our problem, we used the classic Fourth-Order Runge-Kutta algorithm, the explicit version. Using an implicit algorithm imposes more than one unknown per row, which requires implementing a matrix decomposition solver and then solving the linear system. Our initial intention was to measure the complexity of implementing a stiff solver in VHDL, and we wanted to make the algorithm as simple as possible. So the performance was not our concern for this case.

We first implemented in software using C language, that prototype as necessary to guarantee the correct and fast implementation of the floating-point operations. From this software version, we generated the graph (Figure 30), which represents the preditor-prey stiff problem.

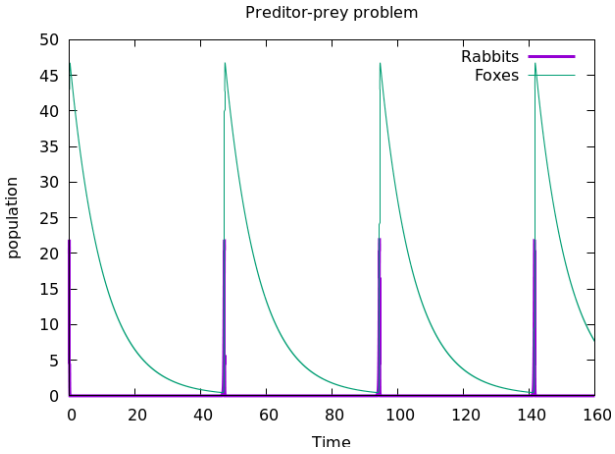

Figure 30 – Preditor-Prey Stiff Problem

Using floating-point in VHDL is not a simple task. It does require advanced knowledge and some expertise with timing and verification in FPGA. In the first design, we implemented a combinational circuit to process the 4 stages of the implicit Runge-Kutta with double precision by David Bishop's library for floating-point. We used Stratix V FPGA (5SGXEA7K2F40C2), which is the one we have available in the laboratory. We knew that designing a combinatorial circuit was not an efficient approach, but we needed the algorithm to be as simple as possible for managing the complexity of the design.

This design imposed some problems since the beginning, and Intel demands a VHDL code close to their role model to avoid errors like *read during* on RAM, which took us a while to figure out how to solve. Our results showed that the clock frequency was very slow (1.91MHz), and the timing report had negative slacks, which was expected since we were building a combinatorial circuit with a long critical path.

We developed a pipelined version in a second design to fix clock frequency, improve critical path, and fix the negative slacks. Our algorithm pipeline contains 27 stages, and it is enough to improve maximum clock frequency but still requires attention over the timing. According to our timing report, we improved the negative slacks in 26×, and that is when we reached a point where we cannot further improve timing without modifying David Bishop's library. In the TimeQuest Timing Analyzer, we found that the multiplication was causing the negative slacks, we also tried to use the compiler features to improve the design without any modification over the library, but that did not work either. We show the compilation report for both designs in Table 28.

Table 28 – Hardware resources for preditor-prey circuits

	Combinatorial	**Pipeline**
Registers	808 (< 1%)	6,967 (12%)
Logic	83,843 (36%)	49,021 (21%)
DSPs	104 (41%)	64 (25%)
RAM blocks	5 (< 1%)	5 (< 1%)
Maximum Clock Frequency (MHz)	1.91	45.73

As we mentioned earlier, working with floating-point in VHDL is not trivial, which becomes even worse when using double precision. We found that the library does not use pipelines for the math operations by reading the source code. That explains the negative slacks and the low frequency for all of our designs. We could use FloPoCo, a C++ framework responsible for generating a pipelined arithmetic operation given the desired frequency. The framework could solve our problem, but that would require that the generated VHDL be compatible with the VHDL pattern accepted by OpenCL. Given the complexity of such a problem, we gave up on the idea and decided to proceed with our designs using only OpenCL. We have finished this activity in July of 2019.

A.2 Technical issues and learning curve

In July, we started our OpenCL experiments at Paderborn University, but they did not provide any resources for compilation. So we had to compile the designs at Euler Cluster provided by FAPESP (grant 2013/07375-0), we also compiled some designs at our laboratory, but our resources were not competitive to improve compiling time. A regular compilation usually takes 8 hours to generate a design. In our laboratory, we already had some designs taking two full days. That is why we were avoiding our laboratory resources.

In November, Euler had to stop operating because of some technical issues related to the air conditioning. At the same time, our laboratory suffered some external attack erasing the entire content of our hard drive. We had a backup of our source codes, but we still needed to Install and set the Intel OpenCL environment all over again. That takes a long time, and most

of the time, we need to do it more than once, caused by driver errors or poor documentation provided by Intel. That happened during the BEPE internship in Portugal, and there was not anyone who could help set the environment again.

So we needed a new environment where we could run and compile as fast as possible. Our only choice was IL Academic Compute Environment (ACE) by Intel, which has the same board as Paderborn University and provided resources to compile the kernels in their cluster. Since we were working in a new environment, we had a learning curve on submitting the jobs and working with this new architecture.

At Paderborn, we struggled to design any kernel that worked in the hybrid architecture Xeon (CPU) + FPGA. Anything different from the examples would not work. Since ACE has the same architecture, we had to stop implementing and start working on the FPGA documentation. Although we already had some experience with OpenCL, that was insufficient to program this new architecture properly. We had to learn concepts like shared memory, which was the leading cause of our mistakes. Such memories require a lock and unlock from the accelerator to guarantee the integrity of the data, so the FPGA can only access the RAM after locking that slice of the memory. Otherwise, the system will fail or enter into an infinite loop. That kind of management is performed by the host side, which explains why we could generate the bitstream but could not execute them. We realized that we needed to modify the entire host side.

APPENDIX

B

PERFORMANCE RESULTS FOR THE ROSENBROCK WITH MEMORY-BOUND FUNCTIONS

This stage required our previous QR Decomposition algorithm. We had to make some changes to the QR algorithm's memory structure; otherwise, it would not compile due to area restrictions or compiler errors. The first change was on the dot product function, which is now using shift registers instead of loop unrolling. This modification incurs performance penalties, but we did not have block RAMS to spend on the memory replication of the loop unrolling. We also had to improve memory utilization used by the upper triangular matrix R, which can work by using half of the initial memory. We have also implemented another version of QR with padded loops to avoid stalls, and we show the results for both versions in Table 29.

Besides the modification of the memory structure of the QR, we also modified Jacobian and Fexchem algorithms. Since both computations were similar, we merged the computation. That did not incur any performance penalty and allowed us to spare memory resources. According to Table 29, we do not have more memory resources to increase the algorithm in the same FPGA, which requires extrapolation analysis from now on.

Table 29 – Hardware resources for jacc + fexchem + qr

	No stalls	**With stalls**
Registers	321,303 (18%)	343,367 (20%)
Logic	186,415 (44%)	195,998 (46%)
DSPs	210 (14%)	215 (14%)
RAM blocks	2,400 (88%)	2,411 (89%)
Maximum Clock Frequency (MHz)	137.5	145.0

Table 30 – Time execution for jacc + fexchem + qr

	No stalls	**With stalls**
Send (μs)	23	17
Execution (μs)	237,658	115,650
Receive (μs)	12	8
Total (μs)	237,693	115,675

As we can see in Table 30, padding the QR algorithm loops are $2\times$ slower than the original algorithm without any modification. According to the report, our loop padding inserted some dependencies on the matrices Q and R, which forced serial execution. Since we were computing all the matrix columns and not the upper triangular, that does incur the double of computation. We also tried to fully unroll the dot product, which did not fit on the board. We used data parallelism for the dot product with a factor of 6, which is almost a multiple of 47. According to our results, that dropped the performance in 2%.

Comparing the results from Table 30 with our previous published results in (JUNIOR *et al.*, 2020). We realized that this version with the entire Rosenbrock algorithm in the FPGA is almost $2\times$ slower than the computing QR algorithm with the initial variables being communicated, which made us realized that this is one of those situations where communicating is less expensive than computing. We could not fit in a single FPGA the backward substitution, so further research of this Ph.D. project will consider the results' extrapolation if there were more than one FPGA to solve this problem.

APPENDIX

C

STREAMING ROSENBROCK

We have two unrolling factors in the format X.Y, the X factor is related to the dot product unrolling, and the Y factor is related to the fexchem unrolling.

Unrolling Factor	**Time (μs)**
1.2	29,424
1.3	28,714
2.1	28,770
2.2	27,085
2.3	28,041
3.1	30,030
3.3	29,863

Unrolling Factor	**1.2**	**1.3**	**2.1**	**2.2**	**2.3**	**3.1**	**3.3**
Registers	569,992	577,529	578,169	602,456	591,453	596,713	609,987
Logic	274,608	277,652	277,498	291,998	284,051	289,795	295,831
DSPs	230	242	242	278	266	266	290
RAM blocks	2118	2,136	2,141	2278	2,171	2,272	2,302
Frequency (Mhz)	158.17	161.68	147.81	159,93	148.75	145.3125	143.333333334

ANNEX

A

ADAPTED SOURCE CODE FOR BRAMS' ROSENBROCK

Source code 7 – C program for computing sparse linear systems

```
#include <sys/types.h>
#include "spConfig.h"
#include "spmatrix.h"
#include "spDefs.h"
#include <stdio.h>
//#include <stdlib.h>
#include <math.h>
#include <string.h>
#define II_CYCLES_ACCUM 9
#define FACTOR 2
#define ITERATIONS 48/FACTOR
#define FACTOR_FEXCHEM 2
#define FEXCHEM_ITERATIONS 75/FACTOR_FEXCHEM
#define C83 8/3
#define ARRAY_SIZE 47
#define BLOCK 65
#define REACTIONS 128
#define NNZ_DW 222

 int initialize_variables_(double *rk_fortran, double *scp_fortran);
 void fexchem_(int nreactions, int nspecies, double *local_rk, double *local_y, double *
    local_dlr);
 void solve_linear_(int *nspecies, double dlr[47], double result[47], int *factorize);


 extern struct crk_{
  double dataRk[65*128];
 } crk_;

 int initialize_variables_(double *rk_fortran, double *scp_fortran);
 void fexchem_(int nreactions, int nspecies, double *local_rk, double *local_y, double *
    local_dlr);
 void solve_linear_(int *nspecies, double dlr[47], double result[47], int *factorize);

double dataA[ARRAY_SIZE*ARRAY_SIZE] = {0.008356, -0.000000, 0.000020, -0.000020,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
```

```
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    0.000001, -0.000000, -0.000001, -0.000000, -0.000000, -0.000000, -0.000000, 0.000000,
     -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, 0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
34              -0.000000, 0.008333, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
35              0.007629, -0.000000, 0.015962, -0.007629, 0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, 0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, 0.000000, 0.000000, 0.000000, -0.000000, -0.000000,
    0.000000, 0.000000, 0.000000, -0.000000,
36              0.000013, -0.000000, -0.000000, 0.008346, -0.000013, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, 0.000000, 0.000000, -0.000000,
     -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, 0.000000, 0.000000, -0.000000,
     0.000000, -0.000000, -0.000000,
37              -0.000000, -0.000000, 0.032613, -0.046712, 0.155322, -0.018533, -0.000000,
    -0.000073, -0.000000, -0.000000, -0.000000, -0.000030, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000026,
    -0.000000, -0.000000, -0.000000, 0.000138, 0.095613, 0.000000, 0.000023, 0.000045,
    -0.000000, -0.087712, -0.000000, 0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, 0.000000, 0.000000, 0.000000, -0.000000, -0.000000, 0.000000,
    -0.000047, -0.095742, -0.000001,
38              -0.000000, -0.000000, -0.000000, -0.023654, -0.023654, 0.031988, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
39              -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, 0.008333,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
40              -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
     0.008333, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
41              -0.000000, -0.000000, -0.000000, -0.048183, -0.000000, -0.000000, -0.000000,
     -0.000000, 0.056517, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.048183,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
```

```
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
42              -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
     -0.000000, -0.000000, 0.008333, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, 0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
43              -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
     -0.000000, -0.000000, -0.000000, 0.008333, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
44              -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
     -0.000000, -0.000000, -0.000000, -0.000000, 0.008333, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, 0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
45              -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
     -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, 0.008333, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
46              -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
     -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, 0.008333,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
47              -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
     -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    0.008333, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
48              -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
     -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, 0.008333, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
49              -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
     -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, 0.008333, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
50              -77436.298921, -0.000000, -0.138916, 0.162288, -0.023372, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.065714, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, 77443.051476, -0.000000, -0.131427,
```

```
    -1.839983, -0.000000, -0.000000, -0.000000, -6.036983, 6.571366, -0.000000,
    -0.328568, -0.000000, -0.000000, -0.871035, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.985705,
51          -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
     -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -719149154.597408, 840256257.842559,
    -242214206.473636, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
52          0.029333, 0.000000, 0.006097, 0.171589, -0.000533, -0.000000, -0.006097,
    -0.171276, 0.000000, 0.000000, -0.000000, 0.124019, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, 14.142115, -1.261046, 0.229139, 0.001837,
     0.716671, 0.167367, 11.272654, 0.007067, 0.443049, 0.299556, -0.005173, 0.015943,
    0.000000, 0.000220, 0.000000, 0.000000, -0.006292, -0.000000, -0.229139, -0.631153,
    -0.171600, -11.272654, -0.000020, -0.006640, -0.000000, -0.393519, -0.000000,
    -0.003725,
53          0.000905, -0.000000, 0.010573, 0.010287, 0.000000, -0.000000, -0.000000,
    -0.000000, -0.020860, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.011478, 0.040672,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, -0.000000,
    0.000000, 0.000000, 0.000000, 0.000000,
54          -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
     -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, 0.000000, -0.000000, 0.008333,
     -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000,
55          -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
     -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, 0.000000, -0.000000,
    -0.000000, 0.008333, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
56          -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
     -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, 0.000000, -0.000000,
    -0.000000, -0.000000, 0.008333, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
57          0.000002, -0.000000, -0.000000, -0.000000, 0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000001, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000001, -0.000000,
    -0.000000, -0.000000, -0.000000, 0.008335, -0.000000, -0.000000, -0.000001,
    -0.000001, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
58          0.000005, -0.000000, -0.000000, -0.000000, 0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000002, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000001, -0.000002,
    -0.000000, -0.000000, -0.000000, -0.000002, 0.008339, -0.000000, -0.000005,
    -0.000000, -0.000000, -0.000002, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000001, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000001, -0.000000, -0.000001,
```

```
59              -0.000000, -0.000000, -0.000000, -0.000000, 0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, 0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, 0.008333, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
60              -0.000000, -0.000000, -0.000000, -0.000000, 0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, 0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, 0.008333,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
61              -0.000000, -0.000000, -0.000000, -0.000000, 0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, 0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    0.008333, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
62              -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
     -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, 0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, 0.008333, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
63              0.000000, -0.000000, -0.000000, -0.000000, 0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, 0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, 0.008333, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
64              -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
     -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, 0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, 0.008333, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
65              0.000000, -0.000000, -0.000000, -0.000215, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, 0.008549, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000215, -0.000000, -0.000000,
66              -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
     -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, 0.008333, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
67              -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
     -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
```

```
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, 0.008333,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
68              -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
     -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    0.008333, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
69              -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
     -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, 0.008333, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
70              -0.000000, -0.000000, 0.009451, -0.009451, 0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.009451,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.009451,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, 0.017785, 0.000000, 0.000000, 0.000000, -0.000000, -0.000000,
    0.000000, 0.000000, 0.000000, -0.000000,
71              -0.000000, -0.000000, 0.005308, -0.004859, 0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.003942,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000159,
    -0.001759, -0.002895, -0.000181, -0.000449, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000479, 0.013207, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, 0.000000, -0.000000, -0.000690,
72              -0.000000, -0.000000, 0.008437, -0.008437, 0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.008437,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.011801,
    -0.003554, -0.000440, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, 0.016770, -0.000000, -0.000000,
    -0.000000, -0.000000, 0.000000, -0.000000, -0.000000,
73              -0.000000, -0.000000, 0.004870, -0.004125, 0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.004125,
    -0.000000, -0.000000, -0.000000, -0.001842, -0.000000, -0.000000, -0.002951,
    -0.000000, -0.000000, -0.002214, -0.000745, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, 0.013203, -0.000000,
    -0.000000, -0.000000, 0.000000, -0.000000, -0.000000,
74              -0.000000, -0.000000, -0.000000, 0.288037, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, 0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.030734, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.288037, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, 0.296370,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
75              12.694951, -0.000000, -0.000000, 0.518466, -0.000000, -0.000000, -0.518466,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -12.694951, -57.897810,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -71.111227, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    2908.112258, -2836.992698, -0.000000, -0.000000, -0.000000,
```

```
76              -0.000000, -0.000000, 0.004870, -0.004632, 0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.004632,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.010080, -0.000238, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, 0.013203, 0.000000, -0.000000, -0.000000,
77              -0.000000, -0.000000, 0.014853, 0.113279, 0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.001832,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000869,
    -0.001094, -0.000000, -0.001288, -0.000000, -0.128132, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.011605, 0.000000, 0.000000, 0.000000, -0.000000, -0.000000,
    0.000000, 0.149903, 0.000000, -0.000381,
78              -0.000000, -0.000000, 0.004870, -0.008844, 0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000896,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.001140,
    -0.004927, -0.001844, -0.000000, -0.000896, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, 0.000000, 0.013203, -0.000000,
79              -0.000000, -0.000000, 0.004870, -0.004870, 0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000, -0.000000,
    -0.000000, -0.000000, 0.000000, -0.000000, 0.013203
80              };
81
82 int nnz_elements(int nspecies){
83     int cont = 0;
84     for (int i = 0; i < nspecies; i++){
85         for (int j = 0; j < nspecies; j++){
86             if (dataA[i * nspecies + j] != 0 && dataA[i * nspecies + j] != -0){
87                 cont++;
88             }
89         }
90     }
91 }
92
93 char* sp_matrix;
94 void solve_linear_(int *nspecies, double dlr[47], double result[47], int *factorize){
95     int error;
96     double temp[47 * 47], *temp_ptr[47*47], *temp2, *result_ptr;
97     int cont = 0;
98 if (*factorize == 1){
99     sp_matrix = spCreate(47, 0, &error);
100    spClear(sp_matrix);
101
102    for (int i = 0; i < *nspecies; i++){
103        for (int j = 0; j < *nspecies; j++){
104            if (dataA[i * *nspecies + j] > 0 || dataA[i * *nspecies + j] < 0){
105                temp2 = spGetElement (sp_matrix, i+1, j+1);
106        if (temp2 == NULL) printf("NULL\n");
107                *temp2 = dataA[i * *nspecies + j];
108            }
109        }
110    }
111
```

```
112     error = spOrderAndFactor (sp_matrix, NULL, 0.0000001, 0, 1);
113 }
114     spSolve(sp_matrix, dlr, result);
115 }
116 
117 void fexchem_(int nreactions, int nspecies, double *local_rk, double *local_y, double *
    local_dlr) {
118     double local_w[128];
119     const int y_indices_fexchem[128][2] =   {{3, 47}, {0, 47}, {0, 47}, {6, 47}, {7, 47},
     {8, 47}, {4, 47}, {4, 47},
120                                             {1, 47}, {27, 47}, {27, 47}, {28, 47}, {33,
    47}, {34, 47}, {29, 47}, {30, 47},
121                                             {31, 47}, {17, 47}, {17, 0}, {18, 47}, {18,
    47}, {18, 47}, {0, 19}, {0, 20},
122                                             {19, 20}, {1, 19}, {20, 20}, {20, 20}, {17,
    2}, {17, 3}, {17, 3}, {19, 2},
123                                             {19, 3}, {19, 4}, {20, 2}, {20, 3}, {8, 47},
     {20, 4}, {19, 6}, {19, 7},
124                                             {19, 8}, {0, 2}, {0, 3}, {2, 2}, {4, 2}, {4,
     3}, {4, 3}, {5, 47},
125                                             {4, 4}, {19, 47}, {19, 9}, {11, 19}, {25,
    17}, {30, 17}, {21, 19}, {22, 19},
126                                             {23, 19}, {24, 19}, {25, 19}, {26, 19}, {27,
     19}, {28, 19}, {29, 19}, {30, 19},
127                                             {33, 19}, {34, 19}, {32, 19}, {31, 19}, {27,
     4}, {28, 4}, {30, 4}, {26, 4},
128                                             {24, 4}, {25, 4}, {32, 4}, {24, 0}, {25, 0},
     {30, 0}, {32, 0}, {41, 3},
129                                             {41, 20}, {42, 3}, {42, 47}, {42, 0}, {44,
    3}, {32, 47}, {37, 2}, {38, 2},
130                                             {39, 2}, {40, 2}, {43, 2}, {44, 2}, {45, 2},
     {37, 20}, {38, 20}, {39, 20},
131                                             {40, 20}, {43, 20}, {44, 20}, {45, 20}, {37,
     37}, {38, 37}, {39, 37}, {40, 37},
132                                             {43, 37}, {44, 37}, {45, 37}, {38, 44}, {39,
     44}, {40, 44}, {43, 44}, {44, 44},
133                                             {45, 44}, {45, 45}, {45, 45}, {37, 4}, {38,
    4}, {39, 4}, {40, 4}, {43, 4},
134                                             {44, 4}, {45, 4}, {46, 20}, {46, 37}, {46,
    44}, {46, 46}, {46, 2}, {46, 4}};
135 
136     const int index_w[605] = {1, 2, 17, 18, 22, 23, 41, 42, 75, 76, 77, 78, 83, 98, 8,
    25, 26, 27, 75, 76, 0,
137                               3, 6, 28, 29, 31, 34, 41, 43, 44, 45, 86, 87, 88, 89, 90,
    91, 92, 126, 0, 4, 5,
138                               7, 16, 28, 29, 30, 32, 33, 34, 35, 36, 37, 38, 40, 41, 42,
    43, 44, 46, 47, 48,
139                               67, 70, 74, 78, 79, 81, 84, 85, 86, 87, 88, 89, 90, 91, 92,
     106, 114, 115, 116,
140                               117, 118, 119, 120, 121, 126, 127, 5, 6, 7, 30, 33, 37, 39,
     42, 44, 45, 46, 47,
141                               48, 66, 68, 69, 70, 71, 72, 73, 74, 115, 116, 117, 118,
    119, 120, 121, 127, 46,
142                               47, 3, 31, 38, 81, 4, 32, 37, 39, 68, 69, 70, 71, 5, 35,
    36, 40, 50, 50, 9, 10,
143                               11, 15, 51, 52, 56, 60, 63, 68, 70, 75, 76, 77, 78, -1, -1,
     1, 2, 6, -1, 9, 15,
144                               75, 76, 78, 0, 2, 7, 17, 18, 19, 20, 28, 29, 30, 52, 53,
    76, 1, 19, 20, 21, 3, 4,
```

```
                              5, 8, 12, 13, 21, 22, 23, 24, 25, 31, 32, 33, 34, 37, 38,
    39, 40, 49, 50, 51, 52,
                              54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 75,
     76, 77, 78, 83, 5,
                              10, 11, 12, 13, 15, 16, 22, 23, 24, 25, 26, 27, 33, 34, 35,
     36, 37, 49, 50, 51,
                              52, 56, 59, 60, 63, 65, 66, 68, 70, 75, 76, 77, 78, 80, 82,
     86, 87, 88, 89, 90,
                              91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103,
    104, 105, 106, 107, 108,
                              109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120,
     121, 122, 123, 54,
                              75, 55, 75, 56, 52, 57, 72, 75, 76, 89, 103, 109, 118, 52,
    58, 73, 76, 59, 71,
                              79, 80, 81, 82, 83, 9, 10, 12, 15, 52, 56, 60, 64, 65, 66,
    68, 74, 75, 76, 78,
                              86, 87, 88, 89, 91, 92, 100, 101, 102, 103, 104, 105, 106,
    107, 108, 109, 111,
                              112, 114, 115, 116, 117, 118, 120, 121, 123, 11, 13, 16,
    53, 56, 61, 63, 65, 69,
                              70, 75, 77, 87, 88, 91, 92, 101, 102, 105, 106, 107, 108,
    111, 112, 114, 116,
                              117, 120, 121, 14, 16, 56, 62, 63, 65, 70, 75, 87, 88, 92,
    101, 102, 106, 107,
                              108, 111, 112, 114, 116, 117, 121, 15, 52, 53, 56, 63, 66,
    70, 73, 76, 77, 87,
                              89, 90, 91, 101, 103, 104, 105, 107, 109, 110, 111, 116,
    118, 119, 120, 16, 67,
                              74, 79, 87, 89, 90, 92, 99, 106, 112, 113, 114, 121, 66,
    74, 78, 84, 85, 12, 64,
                              93, 13, 65, 77, 94, 95, 96, 97, 98, 122, 56, 75, 76, 77,
    78, 75, 77, 98, 105,
                              107, 108, 109, 111, 112, 11, 13, 54, 64, 75, 76, 86, 87,
    91, 93, 100, 101, 102,
                              103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 115, 116,
     120, 123, 124, 14,
                              55, 56, 65, 67, 75, 87, 94, 101, 107, 116, 57, 88, 95, 102,
     108, 117, 58, 89, 96,
                              103, 109, 118, 59, 71, 79, 80, 59, 81, 82, 83, 82, 90, 97,
    104, 110, 119, 14, 15,
                              61, 62, 63, 65, 69, 70, 75, 76, 77, 78, 84, 85, 91, 98,
    105, 107, 108, 109, 110,
                              111, 112, 120, 124, 72, 73, 92, 99, 106, 112, 113, 114,
    121, 52, 59, 63, 65, 66,
                              70, 74, 76, 87, 91, 101, 105, 107, 111, 116, 120, 122, 123,
     124, 125, 126, 127};

    const double const_w[296] = {0.17307, 0.01833, 0.001, 2.0, 0.65, 0.7, 2.0, 2.0, 2.0,
    0.1053, 0.4, 0.7, 0.91541,
                                 0.847, 0.95115, 1.81599, 0.3244, 0.75, 1.74072, 0.35,
    2.0, 0.71893, 0.4, 0.3, 0.91567,
                                 0.91924, 0.01, 8.78E-4, 1.01732, 1.33723, 0.3512, 0.36,
    0.64728, 0.13, 0.0, 0.0, 0.0,
                                 0.20842, 0.05409, 0.05, 0.04, 0.09, 0.35, 2.0, 2.0, 0.7,
     0.02, 0.999122, 0.65,
                                 0.5507500000000001, 0.39435, 0.28, 0.20595, 0.036, 0.65,
     2.0, 0.96205, 0.7583, 2.0,
                                 2.0, 0.28, 0.12793, 0.10318, 0.51208, 0.02915, 0.28107,
    0.63217, 0.23451, 0.3, 0.28441,
```

```
175                                 0.08, 0.02, 0.74265, 0.847, 0.95115, 0.12334, 0.18401,
    0.66, 0.98383, 1.02767, 0.82998,
176                                 0.6756, 0.48079, 0.50078, 0.506, 0.07566, 0.17599, 0.5,
    0.8129, 0.04915, 0.25928,
177                                 0.043, 0.03196, 0.91868, 0.37388, 0.37815, 0.48074,
    0.24463, 0.42729, 0.1067, 1.06698,
178                                 0.02, 0.06517, 0.05, 0.0014, 0.35, 0.02915, 0.57839,
    0.4, 0.4829, 0.9, 0.7, 0.03002,
179                                 1.3987, 0.606, 0.05848, 0.23419, 1.33, 0.80556, 1.42894,
     1.09, 0.95723, 0.88625, 0.076,
180                                 0.68192, 0.34, 0.03432, 0.13414, 0.353, 0.03142,
    1.40909, 0.686, 0.03175, 0.2074,
181                                 0.96205, 0.2, 0.08173, 0.06253, 0.07335, 0.05265,
    0.51468, 0.15692, 0.33144, 0.42125,
182                                 0.07368, 1.01182, 0.5607, 0.46413, 0.08295, 0.41524,
    0.71461, 0.68374, 0.06969,
183                                 0.42122, 0.925, 0.33743, 0.43039, 0.02936, 0.9185, 0.8,
    0.03498, 0.00853, 0.37591,
184                                 0.00632, 0.07377, 0.54531, 0.0522, 0.37862, 0.09673,
    0.03814, 0.09667, 0.18819,
185                                 0.06579, 0.0219, 0.10822, 0.217, 0.62978, 0.02051,
    0.3474, 0.13255, 0.00835, 0.83081,
186                                 0.21863, 0.8947, 0.91741, 0.39754, 0.07583000000000006,
    0.03407, 0.45463, 2.06993,
187                                 0.0867, 0.07976, 0.56064, 1.99461, 0.15387, 0.06954,
    0.78591, 1.99455, 0.10777,
188                                 0.03531, 0.6116, 2.81904, 0.03455, 0.6, 0.08459, 0.153,
    0.04885, 0.18401, 0.6756,
189                                 0.66562, 2.0, 1.25, 0.25928, 0.7189300000000001, 0.6,
    0.7, 0.10149, 1.00524, 1.00524,
190                                 1.00524, 1.00524, 0.80904, 1.00524, 0.00878, 0.15343,
    0.15, 0.10788, 0.11, 0.08143,
191                                 0.20595, 0.17307, 0.13684, 0.4981, 0.49922, 0.494,
    0.09955, 0.48963, 0.03795, 0.65,
192                                 0.13966, 0.03, 0.09016, 0.78134, 2.0, 0.9861, 0.43969,
    0.5148, 0.50078, 0.506, 1.66702,
193                                 0.51037, 0.09731, 0.9191, 0.87811, 0.40341, 0.09815,
    0.91813, 0.99615, 0.99172,
194                                 0.91006, 1.02529, 0.00276, 0.93968, 0.98, 0.69622,
    0.51419, 0.05413, 0.38881, 0.05705,
195                                 0.17, 0.2746, 0.7, 0.90468, 0.94046, 1.94179, 0.96825,
    0.93768, 2.0, 2.0, 0.15,
196                                 0.10318, 0.10162, 0.09333, 0.1053, 0.13, 0.13007,
    0.02563, 0.1337, 0.02212, 0.11306,
197                                 0.01593, 0.16271, 0.01021, 2.0, 1, -1};
198
199     const int index_const_w[605] = {295, 295, 294, 295, 295, 295, 295, 295, 295, 295,
    295, 295, 295, 0, 295,
200                                    295, 294, 294, 1, 2, 294, 294, 294, 295, 294, 295,
    295, 295, 3, 295, 294,
201                                    295, 295, 295, 295, 295, 295, 295, 295, 295, 294, 4,
    294, 294, 294, 295,
202                                    295, 295, 294, 294, 295, 294, 5, 294, 294, 294, 295,
    6, 7, 295, 294, 8,
203                                    294, 9, 10, 11, 295, 295, 295, 294, 294, 12, 294, 13,
     14, 294, 15, 16, 17,
204                                    294, 294, 294, 294, 294, 294, 18, 294, 294, 19, 295,
    295, 294, 295, 295,
205                                    294, 294, 295, 295, 295, 294, 20, 21, 295, 295, 295,
    295, 295, 295, 22,
```

```
206                                 295, 295, 295, 295, 295, 295, 295, 295, 294, 295,
    295, 294, 295, 294, 295,
207                                 294, 23, 295, 294, 294, 24, 294, 295, 294, 295, 295,
    295, 294, 294, 294,
208                                 294, 25, 295, 26, 27, 294, 28, 294, 29, 30, 31, 32,
    33, 34, 35, 294, 294,
209                                 294, 36, 294, 37, 38, 39, 40, 294, 294, 294, 295,
    295, 294, 294, 295, 295,
210                                 295, 295, 295, 41, 294, 295, 295, 295, 294, 294, 42,
    43, 294, 294, 44, 295,
211                                 294, 295, 295, 295, 295, 295, 294, 45, 295, 295, 295,
     295, 295, 295, 46,
212                                 295, 295, 47, 295, 295, 295, 295, 295, 295, 295, 48,
    49, 295, 295, 50, 51,
213                                 52, 53, 294, 54, 55, 294, 294, 56, 57, 294, 294, 295,
     295, 294, 58, 59,
214                                 294, 295, 295, 294, 295, 294, 294, 294, 60, 61, 62,
    294, 63, 64, 65, 294,
215                                 66, 67, 68, 69, 70, 295, 71, 294, 72, 294, 73, 74,
    75, 76, 295, 295, 295,
216                                 295, 295, 295, 295, 77, 78, 294, 294, 79, 80, 81, 82,
     83, 84, 294, 85, 86,
217                                 294, 87, 294, 88, 294, 294, 294, 89, 90, 295, 294,
    295, 91, 295, 92, 295,
218                                 93, 295, 295, 295, 94, 95, 96, 97, 98, 295, 295, 295,
     295, 295, 295, 99,
219                                 100, 294, 101, 294, 295, 295, 294, 102, 103, 104,
    295, 105, 106, 107, 295,
220                                 108, 109, 110, 111, 294, 112, 113, 114, 115, 116,
    117, 118, 119, 120, 294,
221                                 121, 122, 123, 124, 125, 126, 127, 128, 294, 129,
    130, 131, 132, 133, 294,
222                                 295, 134, 135, 294, 136, 295, 137, 138, 295, 139,
    140, 141, 142, 143, 144,
223                                 145, 146, 147, 148, 149, 150, 151, 152, 153, 154,
    155, 156, 157, 158, 295,
224                                 159, 160, 295, 161, 162, 163, 164, 165, 166, 167,
    168, 169, 170, 171, 172,
225                                 173, 174, 175, 176, 177, 178, 295, 179, 295, 180,
    181, 182, 183, 184, 185,
226                                 186, 187, 188, 189, 190, 191, 192, 193, 194, 195,
    196, 197, 198, 199, 200,
227                                 201, 202, 295, 295, 203, 294, 204, 205, 206, 207,
    294, 208, 209, 210, 211,
228                                 212, 213, 214, 215, 294, 295, 295, 295, 294, 295,
    295, 216, 217, 218, 219,
229                                 220, 221, 222, 223, 224, 225, 226, 227, 228, 229,
    230, 231, 232, 233, 234,
230                                 235, 236, 294, 237, 294, 238, 239, 240, 295, 241,
    242, 295, 243, 244, 295,
231                                 295, 295, 245, 295, 246, 247, 248, 294, 249, 250,
    295, 251, 252, 295, 294,
232                                 294, 294, 253, 254, 294, 255, 256, 295, 257, 258,
    259, 260, 295, 295, 295,
233                                 295, 295, 294, 295, 295, 295, 295, 295, 261, 294,
    295, 295, 262, 295, 295,
234                                 295, 263, 295, 295, 295, 295, 295, 294, 264, 294,
    294, 265, 266, 294, 267,
235                                 268, 269, 270, 271, 295, 294, 272, 295, 273, 295,
    295, 295, 295, 274, 295,
```

```
236                                     275, 295, 276, 294, 295, 295, 295, 295, 277, 278,
    295, 279, 280, 281, 282,
237                                     294, 283, 294, 284, 285, 286, 287, 288, 289, 290,
    291, 292, 295, 295, 295,
238                                     293, 295, 295};
239
240     const int index_w_per_chem[47] = {14, 6, 19, 49, 29, 2, 4, 8, 4, 1, 1, 15, 1, 1, 3,
    1, 5, 13, 4, 42, 74, 2,
241                                       2, 1, 9, 4, 7, 41, 29, 22, 26, 14, 5, 3, 9, 5, 9,
    28, 11, 6, 6, 4, 4, 6,
242                                       25, 9, 22};
243
244     for (int y = 0; y < 128; y++) {
245         local_w[y] = local_rk[y] * local_y[y_indices_fexchem[y][0]] * local_y[
    y_indices_fexchem[y][1]];
246     }
247
248     int index_w_begin = 0, index_w_accum = 0;
249     for (int y = 0; y < 47; y++) {
250         unsigned short w_per_chem = index_w_per_chem[y];
251         double shift_reg_chem[II_CYCLES_ACCUM + 1];
252
253         for (int i = 0; i < II_CYCLES_ACCUM + 1; i++) {
254             shift_reg_chem[i] = 0;
255         }
256
257         for (unsigned short z = 0; z < FEXCHEM_ITERATIONS; z++) { //74 div 2 == 37
258             double accum_chem = 0;
259
260             for (unsigned short i = 0; i < FACTOR_FEXCHEM; i++) {//data-parallelism
261                 unsigned short index = z * FACTOR_FEXCHEM + i;
262
263                 accum_chem += ((index < w_per_chem) ? local_w[index_w[index_w_begin]] *
264                                                       const_w[index_const_w[index_w_begin
    ]] : 0);
265                 index_w_begin++;
266             }
267             shift_reg_chem[II_CYCLES_ACCUM] = shift_reg_chem[0] + accum_chem;
268
269             for (int i = 0; i < II_CYCLES_ACCUM; i++) {
270                 shift_reg_chem[i] = shift_reg_chem[i + 1];
271             }
272         }
273
274         double sum_chem = 0;
275
276         for (int i = 0; i < II_CYCLES_ACCUM + 1; i++) {
277             sum_chem += shift_reg_chem[i];
278         }
279
280         local_dlr[y] = sum_chem;
281         index_w_accum += w_per_chem;
282         index_w_begin = index_w_accum;
283     }
284 }
285
286 int initialize_variables_(double *rk_fortran, double *scp_fortran) {
287     int i = 0, size = ARRAY_SIZE;
288     double rke, scpe;//, dataRk[BLOCK * REACTIONS], dataScp[BLOCK * ARRAY_SIZE];
289     FILE *rk, *scp;
```

```

    rk = fopen("rk.dat", "rb");
    scp = fopen("y.dat", "rb");
    if (rk == NULL) {
        printf("Error RK!");
        exit(1);
    }

    if (scp == NULL) {
        printf("Error SCP!");
        exit(1);
    }

    i = 0;
    while(fread(&rke, sizeof(double), 1, rk) > 0){
    rk_fortran[i] = rke;
        i++;
    }

    i = 0;
    while(fread(&scpe, sizeof(double), 1, scp) > 0){
        scp_fortran[i] - scpe;
        i++;
    }

    fclose(rk);
    fclose(scp);
    return 0;
}
```

Source code 8 – Fortran 90 program for computing rosenbrock Method

```
program rodas3

 !USE solve_sparse, ONLY: Solve_linear      ! Subroutine, OUT: spack_2d(ijk,inob)%DLk1

 implicit none

 INTEGER :: i, j, ijk, nreactions = 128, nspecies = 47, block_end = 65
 integer, dimension (47,47) :: matrix
 double precision, dimension (65*128) :: dataRk
 double precision, dimension (65*47) :: dataScp
 double precision, dimension (47) :: dlr1, dlr2, dlr3, dlr4, result1, result2, result3,
    result4
 double precision, dimension (128) :: local_rk
 double precision, dimension (48) :: local_scp, local_scp_new
 double precision dt_chem_i
 real :: startTime, stopTime, increment = 0
 common /crk/ dataRk
 common /cscp/ dataScp

 CALL initialize_variables(dataRk, dataScp)
 !call exit(1)
call cpu_time(startTime)
 DO ijk=1,block_end
     DO j=1,nreactions
          local_rk(j) = dataRk(0 * block_end + j)
     ENDDO

     local_scp(48) = 1
```

```
28      local_scp_new(48) = 1
29      DO j=1,nspecies
30           local_scp(j) = dataScp(0 * block_end + j)
31      ENDDO
32
33      ! Stage 1
34      CALL fexchem(nreactions, nspecies, local_rk, local_scp, dlr1)
35 !call cpu_time(stopTime)
36 !increment = increment + (stopTime - startTime)
37
38 !write(*, '(A, F8.6)') 'Elapsed time, s : ',  (stopTime - startTime)
39 !print *, "passou fexchem"
40      !CALL Solve_linear(nspecies, matrix, dlr1, result1)
41 !call cpu_time(startTime)
42      CALL Solve_linear(nspecies, dlr1, result1, 1)
43 !call cpu_time(stopTime)
44 !increment = increment + (stopTime - startTime)
45 !PRINT *, "Passou solve linear 1"
46
47      ! Stage 2
48 !call cpu_time(startTime)
49      dt_chem_i = 1.0d0/0.5
50      DO i=1,nspecies
51           dlr2(i) = (4.0D0 * dt_chem_i) * result1(i) + dlr1(i)
52      ENDDO
53 !call cpu_time(stopTime)
54 !increment = increment + (stopTime - startTime)
55 !print *, "Update dlr2"
56      !CALL Solve_linear(nspecies, matrix, dlr2, result2)
57 !call cpu_time(startTime)
58      CALL Solve_linear(nspecies, dlr2, result2, 0)
59 !call cpu_time(stopTime)
60 !increment = increment + (stopTime - startTime)
61 !print *, "solve 2"
62      !Stage 3
63 !call cpu_time(startTime)
64      DO j=1,nspecies
65           local_scp_new(j) = local_scp(j) + 2.0D0* result1(j)
66
67           IF (local_scp_new(j) .LT. 0.0) THEN
68                local_scp_new(j) = 0.0
69                result1(j) = 0.5D0 * (local_scp_new(j) - local_scp(j))
70           ENDIF
71      ENDDO
72 !call cpu_time(stopTime)
73 !increment = increment + (stopTime - startTime)
74 !print *, "scp_new"
75 !call cpu_time(startTime)
76      CALL fexchem(nreactions, nspecies, local_rk, local_scp_new, dlr3)
77 !call cpu_time(stopTime)
78 !increment = increment + (stopTime - startTime)
79 !print *, "dlr3"
80 !call cpu_time(startTime)
81      DO j=1,nspecies
82           dlr3(i) = dlr3(i) + dt_chem_i * ( result1(j) - result2(j))
83      ENDDO
84 !call cpu_time(stopTime)
85 !increment = increment + (stopTime - startTime)
86 !print *, "update dlr3"
87      !CALL Solve_linear(nspecies,matrix, dlr3, result3)
```

```
!call cpu_time(startTime)
     CALL Solve_linear(nspecies, dlr3, result3, 0)
!call cpu_time(stopTime)
!increment = increment + (stopTime - startTime)
!print *, "solve 3"
     ! Stage 4
!call cpu_time(startTime)
     dt_chem_i = 1.0d0/0.5
     DO j=1,nspecies
          local_scp_new(j) = local_scp(j) + 2.0D0 * result1(j) + result3(j)

          IF (local_scp_new(j) .LT. 0.0) THEN
               local_scp_new(j) = 0.0
               result3(j) = (local_scp_new(j) - local_scp(j)) -2.0D0 * result1(j)
          ENDIF
     ENDDO
!call cpu_time(stopTime)
!increment = increment + (stopTime - startTime)
!print *, "scp_new 4"
!call cpu_time(startTime)
     CALL fexchem(nreactions, nspecies, local_rk, local_scp_new, dlr4)
!call cpu_time(stopTime)
!increment = increment + (stopTime - startTime)
!print *, "fexchem dlr 4"
!     call cpu_time(stopTime)
     DO j=1,nspecies
          dlr4(j) =  dlr4(j) + dt_chem_i* (result1(j) - result2(j) - ((8/3) * result3(j))
    )
     ENDDO
!call cpu_time(stopTime)
!increment = increment + (stopTime - startTime)
!print *, "update dlr 4"
!     CALL Solve_linear(nspecies, matrix, dlr4, result4) !matrix solution
!call cpu_time(startTime)
     CALL Solve_linear(nspecies, dlr4, result4, 0) !matrix solution
!call cpu_time(stopTime)
!increment = increment + (stopTime - startTime)
!print *, "resultado"
ENDDO
call cpu_time(stopTime)
write(*, '(A, F8.6)') 'Elapsed_time : ',  (stopTime - startTime)
end program
```


ICMC
USP
SÃO CARLOS