UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

Um framework para coprojeto de hardware/software para o módulo da dinâmica do modelo brasileiro de previsão do tempo - BRAMS

Erinaldo da Silva Pereira

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

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Erinaldo da Silva Pereira

Um framework para coprojeto de hardware/software para o
módulo da dinâmica do modelo brasileiro de previsão do
tempo - BRAMS

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

USP – São Carlos
Março de 2019

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

d436f da Silva Pereira, Erinaldo
Um framework para coprojeto de hardware/software
para o módulo da dinâmica do modelo brasileiro de
previsão do tempo - BRAMS / Erinaldo da Silva
Pereira; orientador Eduardo Marques. -- São Carlos,
2018.
200 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, 2018.

1. Framework. 2. coprojeto. 3. FPGA. 4. BRAMS.
5. OpenCL. I. Marques, Eduardo, orient. II. Título.

Erinaldo da Silva Pereira

A framework for the hardware/software codesign for the
dynamic module of the Brazilian model of weather forecast -
BRAMS

Doctoral dissertation submitted to the Institute of
Mathematics and Computer Sciences – ICMC-USP, in
partial fulfillment of the requirements for the degree of
the Doctorate Program in Computer Science and
Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and
Computational Mathematics

Advisor: Prof. Dr. Eduardo Marques

USP – São Carlos
March 2019

*Este trabalho é dedicado a minha mãe, Juranía Maria da Silva,
a mulher que transformou o meu mundo, me deu o seu melhor, me permitiu sonhar e sempre
acreditou no meu potencial.
Te Amo Mãe!*

AGRADECIMENTOS

Agradeço a Deus por ter cuidado de mim em todos os momentos da minha vida, sempre me dando forças para superar as dificuldades ao longo desta caminhada. A Nossa Senhora de Fátima por me abençoar e me conceder no seu dia a graça de ter sido aprovado no concurso para professor. Ao meu anjo da guarda que tem sido a minha espada e o meu escudo nos momentos de dificuldade.

A minha família, em especial minha mãe, Jurânia e aos meus irmãos. Eduardo que se comportou como um pai ao longo da minha vida, e Erivalda. A minha esposa Maria Adelina, nos momentos de dificuldade você foi a rocha que me sustentou e me deu todo o apoio necessário. A Liana minha filha, você se tornou a razão do meu viver, o motivo pelo qual acordo todos os dias buscando ser uma pessoa cada vez melhor.

Ao meu Orientador e Amigo, Eduardo Marques. O senhor viabilizou a realização deste e de outros sonhos, a sua confiança depositada, apoio, conselhos e sugestões, e pela competente orientação o meu eterno Obrigado e Gratidão.

Agradeço de modo especial ao professor Jairo Panetta, ao Luiz Flávio e Saulo Freitas do CPTEC/INPE e ao Professor Alvaro Fazenda, sem o conhecimento que vocês me proporcionaram sobre o BRAMS o meu caminho teria sido ainda mais complicado, o meu Muito Obrigado.

Aos professores Paulo Matias e Vanderlei Bonato pelos conselhos e momentos de orientação, fatores que contribuíram para a conclusão deste trabalho.

Ao meu irmão caçula, Leonan, você também foi fundamental para realização deste trabalho, muito obrigado pelos finais de semana e tempo disponível para me ajudar.

Muito obrigado aos colegas do LCR, de modo especial ao Carlos pela parceria formada ao longo deste trabalho. Carlão sempre disposto a ajudar, André Perina obrigado por me mostrar um caminho diferente na implementação, a Fatemeh pelo carinho e atenção e ao demais que estiveram presentes nos momentos de dificuldades e de alegrias. Aos idealizadores e coordenadores do programa HARP da Intel por disponibilizar uma moderna infraestrutura de computação heterogênea para pesquisa e o suporte fornecido .

Aos funcionários do ICMC, de modo especial Carol da pós-graduação que, direta ou indiretamente, contribuíram significativamente para que este objetivo fosse alcançado. Agradecimento especial a CAPES pelo apoio financeiro. "O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

A todos vocês, Muito Obrigado!

*“A emoção mais antiga e mais forte da humanidade é o medo, e o mais antigo
e mais forte de todos os medos é o medo do desconhecido.”
(H.P. Lovecraft)*

RESUMO

PEREIRA, E. S. **Um framework para coprojetado de hardware/software para o módulo da dinâmica do modelo brasileiro de previsão do tempo - BRAMS**. 2019. 200 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, 2019.

O BRAMS (*Brazilian developments on the Regional Atmospheric Modelling System*) é o sistema utilizado pelo CPTEC/INPE para previsão climática no Brasil. Este projeto de doutorado contribui para a modernização do código desse sistema a partir da implementação e avaliação de um *framework* para coprojetado de hardware/software do módulo da dinâmica do modelo climático BRAMS. Foi conduzido um estudo do código do BRAMS para verificar quais trechos poderiam ser acelerados em hardware. Com isso foram desenvolvidos kernels usando Intel *OpenCL* para serem executados em dispositivos programáveis do tipo FPGA. Este estudo utilizou o suporte e recursos do programa da Intel HARP (*Heterogeneous Architecture Research Platform*), que disponibilizou uma infraestrutura de computação heterogênea com processadores Xeon com um FPGA Arria 10 integrado. Foram conduzidos dois estudos de caso em que os resultados sugerem que é possível portar uma aplicação climática para uma máquina heterogênea que utiliza CPU e FPGA. Porém, para obter um desempenho satisfatório nessa nova arquitetura faz-se necessário domínio dos recursos disponíveis no Intel OpenCL para programar a máquina heterogênea e a aplicação alvo deve possuir uma estrutura de código que favoreça a execução de tais estruturas. Apesar do desempenho com o FPGA Arria 10 não ter sido superior ao do sistema executando apenas em Intel Xeon, o ganho em eficiência de energia justifica a migração do código para esta nova plataforma. Além disso, o framework desenvolvido possibilitará futuras implementações do BRAMS visando uma arquitetura heterogênea como alvo.

Palavras-chave: coprojetado, FPGA, Framework, BRAMS, HARP, OpenCL.

ABSTRACT

PEREIRA, E. S. **A framework for the hardware/software codesign for the dynamic module of the Brazilian model of weather forecast - BRAMS**. 2019. 200 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, 2019.

BRAMS (Brazilian developments on the Regional Atmospheric Modelling System) is the system used by CPTEC/INPE for climate forecast in Brazil. This PhD project contributes to the improvement of the code of this system from implementation and evaluation of a hardware/software codesign framework of the dynamics module of the BRAMS climate model. A study of the source code was conducted to verify what parts can be accelerated with hardware. Kernels were developed using Intel OpenCL and they were executed in programmable devices of the type FPGA. This study used resources of the Intel HARP program (Heterogeneous Architecture Research Platform). HARP provided an infrastructure of heterogeneous computation with Xeon processors including an Arria 10 FPGA integrated. The results from three case studies conducted suggest that it is possible to carry a climate application to a heterogeneous machine that uses CPU and FPGA. However, to obtain a satisfactory performance in this new architecture it is necessary to master the available resources in Intel OpenCL to program the heterogeneous machine and the target application must have a code structure that favors the execution of such structures. Although the performance was not higher than the system running only in CPU, the gain in energy efficiency justifies the migration of the code to this new platform.

Keywords: co-design, FPGA, Framework, BRAMS, HARP, OpenCL.

LISTA DE ILUSTRAÇÕES

Figura 1 – Espaço de coprojetos hardware/software	42
Figura 2 – Exemplo para <i>profiling</i> que não identifica caminhos com maior frequência	45
Figura 3 – Flexibilidade vs. desempenho de classes de processadores	47
Figura 4 – Estrutura de um FPGA	49
Figura 5 – Arquitetura do FPGA Intel Arria 10	50
Figura 6 – Progressão FPGAs da Xilinx em relação capacidade, preço e energia por elemento lógico	51
Figura 7 – Plataforma processador IvyTown Xeon + FPGA Stratix V	54
Figura 8 – Plataforma integrada Intel Xeon + FPGA Arria 10	55
Figura 9 – Intel Xeon Scalable integrado com um FPGA Arria 10	56
Figura 10 – Modelo de programação <i>OpenCL</i>	57
Figura 11 – Fluxo de programação Intel <i>OpenCL</i> para FPGA's da Intel	60
Figura 12 – <i>Pipelines</i> para os modelos de programação <i>Task</i> e <i>NDRange</i>	61
Figura 13 – Interações entre os componentes do BRAMS	63
Figura 14 – Grafo de chamadas do BRAMS com o módulo da química e dinâmica em execução	65
Figura 15 – Arquitetura alvo para particionamento hardware/software.	68
Figura 16 – Arquitetura de um acelerador: processador de propósito geral conectado a uma GPU e FPGA	70
Figura 17 – Arquitetura do sistema	73
Figura 18 – Visão geral do processo de avaliação de uma tecnologia	74
Figura 19 – Publicações relacionadas ao estudo de desempenho do BRAMS	84
Figura 20 – Fluxograma simplificado da metodologia	87
Figura 21 – Diagrama de bloco do <i>kit</i> de desenvolvimento DE2i150	88
Figura 22 – Diagrama de blocos da arquitetura do coprojetos	90
Figura 23 – Diagrama de blocos da placa BittWare S5PH-Q	91
Figura 24 – Fluxograma com as chamadas de execução das funções do módulo da dinâmica	93
Figura 25 – Seleção das funções de interesse para implementação em <i>OpenCL</i>	96
Figura 26 – Stretch - Ambiente de desenvolvimento de software	97
Figura 27 – Arquitetura inicial do projeto (Adaptada)	98
Figura 28 – Diagrama da arquitetura do <i>framework</i>	99
Figura 29 – Descrição do fluxo de conversão de código <i>Fortran</i> para <i>OpenCL</i>	101
Figura 30 – Fluxo de compilação de um código <i>OpenCL</i>	102

Figura 31 – Fluxo de integração e execução de um <i>kernel</i> com a dinâmica do BRAMS .	103
Figura 32 – Diagrama de blocos do módulo de instrumentação do código da dinâmica .	105
Figura 33 – Tela principal do <i>framework</i> proposto	106
Figura 34 – Execução dinâmica dos <i>kernels</i> do módulo da turbulência para o modelo Task	110
Figura 35 – Execução dinâmica dos <i>kernels</i> do módulo da turbulência para o modelo NDRange	113
Figura 36 – Execução dinâmica dos <i>kernels</i> do módulo grad para o modelo Task	115
Figura 37 – Execução dinâmica dos <i>kernels</i> do módulo grad para o modelo NDRange . .	117
Figura 38 – Execução dinâmica dos <i>kernels</i> do módulo advect para o modelo Task . . .	118
Figura 39 – Execução dinâmica dos <i>kernels</i> do módulo advect para o modelo NDRange	120
Figura 40 – Execução dinâmica dos <i>kernels</i> do módulo acoustic para o modelo Task . . .	122
Figura 41 – Execução dinâmica dos <i>kernels</i> do módulo acoustic para o modelo NDRange	123
Figura 42 – Execução dos <i>kernels</i> com melhores tempos do módulo da dinâmica no estudo de caso I	124
Figura 43 – Consumo de energia (J) considerando o TDP	126
Figura 44 – Consumo de energia (J) considerando o TDP	128
Figura 45 – Consumo de energia (J) considerando o TDP	129
Figura 46 – Consumo de energia (J) considerando o TDP	131
Figura 47 – Execução dos <i>kernels</i> com melhores tempos do módulo da dinâmica do estudo de caso II	131
Figura 48 – Eficiência de energia para os <i>kernels</i> executados na Stratix V e na Arria 10 .	134
Figura 49 – Visão do <i>kernel</i> 1 da turbulência em nível de <i>loops</i> e operações de <i>load</i> e <i>store</i>	154
Figura 50 – Visão do <i>kernel</i> 2 da turbulência em nível de <i>loops</i> e operações de <i>load</i> e <i>store</i>	155
Figura 51 – Visão do <i>kernel</i> 3 da turbulência em nível de <i>loops</i> e operações de <i>load</i> e <i>store</i>	156
Figura 52 – Visão do <i>kernel</i> 4 da turbulência em nível de <i>loops</i> e operações de <i>load</i> e <i>store</i>	157
Figura 53 – Visão do <i>kernel</i> 1 do grupo <i>Advect</i> em nível de <i>loops</i> e operações de <i>load</i> e <i>store</i>	158
Figura 54 – Visão do <i>kernel</i> 2 do grupo <i>Advect</i> em nível de <i>loops</i> e operações de <i>load</i> e <i>store</i>	159
Figura 55 – Visão do <i>kernel</i> 3 do grupo <i>Advect</i> em nível de <i>loops</i> e operações de <i>load</i> e <i>store</i>	159
Figura 56 – Visão do <i>kernel</i> 4 do grupo <i>Advect</i> em nível de <i>loops</i> e operações de <i>load</i> e <i>store</i>	160
Figura 57 – Visão do <i>kernel</i> 5 do grupo <i>Advect</i> em nível de <i>loops</i> e operações de <i>load</i> e <i>store</i>	161
Figura 58 – Visão do <i>kernel</i> 5 do grupo <i>Grad</i> em nível de <i>loops</i> e operações de <i>load</i> e <i>store</i>	162
Figura 59 – Visão do <i>kernel</i> 3 do grupo <i>acoustic</i> em nível de <i>loops</i> e operações de <i>load</i> e <i>store</i>	163

Figura 60 – Visão do <i>kernel 4</i> do grupo <i>acustic</i> em nível de <i>loops</i> e operações de <i>load</i> e <i>store</i>	164
Figura 61 – Visão do <i>kernel 5</i> do grupo <i>acustic</i> em nível de <i>loops</i> e operações de <i>load</i> e <i>store</i>	165

LISTA DE QUADROS

Quadro 1 – Trabalhos relacionados ao estudo de desempenho do BRAMS	84
--	----

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Kernel no modelo de programação <i>NDRange</i>	58
Código-fonte 2 – Kernel no modelo de programação <i>Task</i>	58

LISTA DE TABELAS

Tabela 1 – <i>Kernels</i> da turbulência sem cópia de dados para memória local para o modelo Task de programação	109
Tabela 2 – <i>Kernels</i> da turbulência com cópia de dados para memória local no modelo Task de programação	109
Tabela 3 – <i>Kernels</i> da turbulência para o modelo NDRange para 1 CU e WI(20,20) na resolução 35X60X60	111
Tabela 4 – <i>Kernels</i> da turbulência para o modelo NDRange para 2 CU e WI(30,30) na resolução 35X60X60	112
Tabela 5 – <i>Kernels</i> da turbulência para o modelo NDRange para 10 CU e WI(20,20) na resolução 35X60X60	112
Tabela 6 – <i>Kernels</i> da turbulência para o modelo NDRange para 2 CU e WI(50,50) na resolução 35X100X100	113
Tabela 7 – <i>Kernels</i> grad sem cópia para memória local para o modelo de programação Task na resolução 35X60X60	114
Tabela 8 – <i>kernels</i> grad com cópia para memória local para o modelo de programação Task na resolução 35X60X60	115
Tabela 9 – <i>Kernels</i> grad para o modelo de programação NDRange para 4 CU e WI(30,30) na resolução 35X60X60	116
Tabela 10 – <i>Kernels</i> grad para o modelo de programação NDRange para 4 CU e WI(50,50) na resolução 35X100X100	116
Tabela 11 – <i>Kernels</i> advect para o modelo de programação Task sem cópia para memória local na resolução 35X60X60	117
Tabela 12 – <i>Kernels</i> advect para o modelo de programação Task com cópia para memória local na resolução 35X60X60	118
Tabela 13 – <i>Kernels</i> advect para o modelo de programação NDRange para 4 CU e WI(30,30) na resolução 35X60X60	119
Tabela 14 – <i>Kernels</i> advect para o modelo de programação NDRange para 2 CU e WI(50,50) na resolução 35X100X100	120
Tabela 15 – <i>Kernels</i> acoustic para o modelo de programação Task sem cópia para memória local na resolução 35X60X60	121
Tabela 16 – <i>Kernels</i> acoustic para o modelo de programação Task sem cópia para memória local na resolução 35X60X60	121

Tabela 17 – <i>Kernels</i> acoustic para o modelo de programação NDRange para 4 CU e WI(30,30) na resolução 35X60X60	122
Tabela 18 – <i>Kernels</i> acoustic para o modelo de programação NDRange para 4 CU e WI(50,50) e resolução 35X100X100	123
Tabela 19 – <i>Kernels</i> turbulência na resolução 35X60X60	125
Tabela 20 – Eficiência de energia dos <i>kernels</i> da turbulência na Arria 10 em relação a execução na CPU	126
Tabela 21 – <i>Kernels</i> grad – Resolução 35x60x60	127
Tabela 22 – Eficiência de energia dos <i>kernels</i> grad na Arria 10 em relação a execução na CPU	127
Tabela 23 – <i>Kernels</i> advect na resolução 35x60x60	128
Tabela 24 – Eficiência de energia dos <i>kernels</i> grad na Arria 10 em relação a execução na CPU	128
Tabela 25 – <i>Kernels</i> acoustic	130
Tabela 26 – Eficiência de energia dos <i>kernels</i> acoustic na Arria 10 em relação a execução na CPU	130
Tabela 27 – <i>Kernels</i> turbulência executados na Stratix V na resolução 35x60x60	132
Tabela 28 – <i>Kernels</i> grad executados na Stratix V na resolução 35x60x60	132
Tabela 29 – <i>Kernels</i> advect executados na Stratix V na resolução 35x60x60	133
Tabela 30 – <i>Kernels</i> acoustic executados na Stratix V na resolução 35x60x60	133

LISTA DE ABREVIATURAS E SIGLAS

AES	<i>Advanced Encryption Standard</i>
ALM	<i>Adaptive Logic Modules</i>
API	<i>Application Programming Interface</i>
ASAP	<i>ASynchronous Array Processor</i>
ASIC	<i>Application Specific Integrated Circuit</i>
ASIP	<i>Application-Specific Instruction set Processor</i>
BRAMS	<i>Brazilian developments on the Regional Atmospheric Modeling System</i>
BSP	<i>Board Support Package</i>
CI	<i>Circuitos Integrados</i>
COSMO	<i>Consortium for Small-scale Modeling</i>
CPTEC	<i>Centro de Previsão do Tempo e Estudos Climáticos</i>
CPU	<i>Central Processing Unit</i>
CUDA	<i>Compute Unified Device Architecture</i>
DMA	<i>Direct Memory Access</i>
DSEL	<i>Domain-Specific Embedded Language</i>
DSPs	<i>Digital Signal Processors</i>
FPGA	<i>Field-Programmable Gate Array</i>
GPU	<i>Graphics Processing Unit</i>
HARP	<i>Hardware Accelerator Research Program</i>
HDL	<i>Hardware Description Languages</i>
HLS	<i>High Level Synthesis</i>
HPC	<i>High Performance Computing</i>
HPRC	<i>High-Performance Reconfigurable Computing</i>
IDE	<i>Integrated Development Environment</i>
INPE	<i>Instituto Nacional de Pesquisas Espaciais</i>
ISEF	<i>Instruction Set Extension Fabric</i>
ISS	<i>Instruction Set Simulators</i>
LCR	<i>Laboratório de Computação Reconfigurável</i>
LUT	<i>Loop-Up Tables</i>
MPI	<i>Message Passing Interface</i>
PCIe	<i>PCI Express</i>

PPG	Processadores de Propósito Geral
QPI	<i>QuickPath Interconnect</i>
RISC	<i>Reduced Instruction Set Computer</i>
SCC	<i>Stretch C Compiler</i>
SDK	<i>Software Development Kit</i>
SoC	<i>System-on-Chip</i>
TDP	Thermal Design Power
ULAs	Unidades Lógica e Aritmética
ULP	<i>Ultra-Low Power</i>
UPI	<i>Ultra Path Interconnect</i>
WG	<i>Work-Groups</i>
WG	<i>Work-Group</i>
WI	<i>Work-Items</i>
WI	<i>Work-Item</i>

SUMÁRIO

1	INTRODUÇÃO	31
1.1	Contextualização	31
1.2	Justificativa	34
1.3	Motivação	35
1.4	Objetivos	36
1.4.1	<i>Geral</i>	36
1.4.2	<i>Específicos</i>	36
1.5	Organização da tese	37
2	FUNDAMENTAÇÃO TEÓRICA	39
2.1	Considerações iniciais	39
2.2	Coprojeto hardware/software	39
2.3	Técnicas de <i>profiling</i>	44
2.4	Computação Reconfigurável	47
2.4.1	<i>FPGA</i>	48
2.4.1.1	<i>FPGA em computação de alto desempenho</i>	50
2.5	<i>HARP</i>	53
2.6	<i>OpenCL</i>	56
2.6.1	<i>Intel OpenCL SDK</i>	58
2.7	<i>BRAMS</i>	59
2.7.1	<i>Dinâmica do BRAMS</i>	63
3	TRABALHOS RELACIONADOS	67
3.1	Considerações iniciais	67
3.2	Trabalhos relacionados a coprojeto hardware/software	67
3.3	Trabalhos relacionados ao desempenho do BRAMS	75
4	METODOLOGIA	85
4.1	Considerações Iniciais	85
4.2	Descrição das etapas	85
4.2.1	<i>Kit de desenvolvimento DE2i150</i>	87
4.2.2	<i>Usando o FPGA como coprocessador</i>	89
4.2.3	<i>Placa BittWare S5PH-Q</i>	90
4.3	Profile do módulo da dinâmica do BRAMS	92

5	ESPECIFICAÇÃO DO <i>FRAMEWORK</i>	95
5.1	Considerações Iniciais	95
5.2	Código-fonte do módulo da Dinâmica	95
5.3	<i>Framework</i> para coprojeto hardware/software	97
5.3.1	<i>Conversão de código Fortran para OpenCL</i>	99
5.3.2	<i>Compilação do kernel OpenCL</i>	101
5.3.3	<i>Execução dos kernels no HARP</i>	102
5.3.4	<i>Integração dos kernels ao código Fortran da dinâmica do BRAMS</i> .	103
5.3.5	<i>Instrumentação do código da dinâmica para invocar o kernel em</i> <i>OpenCL</i>	104
5.3.6	<i>Framework</i>	106
6	RESULTADOS	107
6.1	Considerações iniciais	107
6.2	Estudos de caso: preparação e execução	107
6.3	Estudo de caso I	108
6.3.1	<i>Kernels da turbulência implementados em Task</i>	108
6.3.2	<i>Kernels da turbulência implementados em NDRange</i>	110
6.3.3	<i>Kernels grad implementados em Task</i>	114
6.3.4	<i>Kernels grad implementados em NDRange</i>	115
6.3.5	<i>Kernels advect implementados em Task</i>	116
6.3.6	<i>Kernels advect implementados em NDRange</i>	119
6.3.7	<i>Kernels acoustic implementados em Task</i>	119
6.3.8	<i>Kernels acoustic implementados em NDRange</i>	121
6.4	Estudo de caso II	123
6.4.1	<i>Kernels da turbulência implementados em NDRange</i>	124
6.4.2	<i>Kernels grad implementados em NDRange</i>	126
6.4.3	<i>Kernels advect implementados em NDRange</i>	127
6.4.4	<i>Kernels acoustic implementados em NDRange</i>	129
6.5	Estudo de caso III	130
6.5.1	<i>Kernels turbulência Stratix V</i>	132
6.5.2	<i>Kernels grad Stratix V</i>	132
6.5.3	<i>Kernels advect Stratix V</i>	133
6.5.4	<i>Kernels acoustic Stratix V</i>	133
6.6	Discussão dos resultados	134
7	CONCLUSÃO	137
7.1	Contribuições e limitações	138
7.2	Lições aprendidas	140
7.3	Trabalhos futuros	141

REFERÊNCIAS	143
APÊNDICE A	
VISÃO DOS KERNELS GERADA DURANTE A COM- PILAÇÃO DO CÓDIGO <i>OpenCL</i>	153
ANEXO A	
CÓDIGOS-FONTE EM FORTRAN DOS KERNELS SE- LECIONADOS E DESENVOLVIDOS EM OPENCL	167
ANEXO B	
DINÂMICA DO BRAMS VERSÃO 1	197

INTRODUÇÃO

1.1 Contextualização

A computação de alto desempenho baseia-se no agrupamento de unidades de processamento, conectadas por meios de comunicação físicos e lógicos operando em conjunto, como se fossem uma única máquina. Contudo, em razão das tendências de crescimento da quantidade de dados processados, o aumento da necessidade por computação de alto desempenho tem forçado mudanças severas nas arquiteturas dos computadores, levando a soluções menos centralizadas.

Desenvolvedores de sistemas eletrônicos, tanto do ambiente industrial como do científico, são desafiados a aumentar de forma vertiginosa a velocidade de tais sistemas. No entanto, a necessidade por sistemas cada vez mais velozes não é uma demanda “nova”. O que mudou ao longo dos anos foram os requisitos e as exigências dos mesmos.

Nesse sentido, nas mais diversas áreas de aplicação, prover alto desempenho não é a única característica a ser atingida, há também a necessidade de um hardware “poderoso” e com baixo consumo de energia. Nesse contexto, arquiteturas baseadas em aceleradores são uma alternativa para balancear o *trade-off* entre consumo de energia, uso de memória, paralelismo a nível de instrução e desempenho (GULATI; KHATRI, 2010).

Com o intuito de prover alto desempenho (ou alta exigência computacional), uma solução comum tem sido aumentar o número de processadores em um sistema. Exemplos de sistemas que adotam essa abordagem são aplicações de simulação física e meteorológicas, modelagem financeira, dentre outros. Esse método é comprovado pela predominância de *clusters* de computadores na lista TOP500 de supercomputadores, compostos por Processadores de Propósito Geral (PPG) e *Graphics Processing Unit* (GPU) (TSOI; LUK, 2010).

Apesar de sua ampla utilização, processadores de propósito geral não são ideais para a maioria das aplicações que executam, pois são projetados para executar aplicações genéricas. Quando a meta é alto desempenho, alterações ou adições que melhoram significativamente o

desempenho podem ser sugeridas e/ou adicionadas nas arquiteturas dos processadores. Entretanto, essas alterações diferem de aplicação para aplicação.

As GPUs têm se popularizado como um meio de computação de alto desempenho devido a alta produtividade, não exigindo que o usuário possua conhecimentos avançados de hardware. Essas unidades de processamento permitem que sejam exploradas características de paralelismo em que cálculos matemáticos podem ser executados de forma mais eficiente do que em PPGs (GULATI; KHATRI, 2010; TSOI; LUK, 2010; THOMA *et al.*, 2015).

O uso de aceleradores de hardware em sistemas computacionais é um recurso que permite o aumento de desempenho e, em muitos casos, a melhora da eficiência computacional em relação ao consumo de energia. Essa é uma preocupação, que tem recebido cada vez mais atenção da comunidade científica (LIU *et al.*, 2009; XU, 2012; HARING *et al.*, 2012). Uma tecnologia que vem sendo adotada cada vez mais para o desenvolvimento de aceleradores de hardware para computação de alto desempenho é o *Field-Programmable Gate Array* (FPGA). Diferente das GPUs e dos PPGs, sua utilização eficiente requer conhecimento avançado de hardware. Dada a flexibilidade de um dispositivo FPGA, é possível obter um hardware acelerador customizado para cada tipo de aplicação. Jones *et al.* (2010) apresentam uma comparação de produtividade no desenvolvimento entre as tecnologias GPU e FPGA.

O uso da tecnologia FPGA no desenvolvimento de aceleradores é muito adequado, dado que é possível obter um hardware customizado para cada tipo de problema, enquanto nas tecnologias GPU e PPG as otimizações são feitas a nível de software por meio de compiladores. Nos últimos anos, a parceria Intel-Altera tem fomentado pesquisas sobre o desenvolvimento de dispositivos híbridos com núcleos de *Central Processing Unit* (CPU) e FPGA no mesmo *kit* de desenvolvimento. Uma das soluções lançadas no mercado foi um processador da família Atom série E600C em conjunto com um FPGA da família Arria II GX. Essa combinação permite o desenvolvimento de sistemas embarcados com alto poder de processamento e baixo consumo de energia, família de processadores classificada como *Ultra-Low Power* (ULP). O FPGA disponibilizado pela Altera para esse processador é um modelo de alto desempenho e consumo otimizado, possuindo mais de 60 mil elementos lógicos (HALFHILL, 2016).

Seguindo uma tendência de mercado e da computação, recentemente a Intel anunciou através de seu CEO, Brian Krzanich, uma plataforma de computação híbrida chamada de *Hiperscalares*. Ela é composta por unidades de processamento contendo uma CPU e FPGA. A ideia é inserir num mesmo *chip* um processador multicore Intel Xeon Phi e um FPGA da Altera. Essa plataforma é voltada a computação de alto desempenho, *datacenters* e outras aplicações.

Aceleradores em hardware estão presentes em vários setores, com o objetivo de realizar tarefas que necessitam de alto poder computacional. Exemplos dessas tarefas são análise de modelos meteorológicos, cálculos para detectar tendências em mercado financeiro, processamento de imagem em alta definição e simulações nas mais diversas áreas do conhecimento. Esses processamentos podem envolver grandes quantidades de cálculos matemáticos complexos,

com uso intensivo do processador (*CPU-bound*), podendo consumir horas ou vários dias para concluir uma execução.

Na área financeira, por exemplo, o banco JP Morgan trabalhou em conjunto com a empresa Maxeler Technologies em um projeto para acelerar cálculos sobre análises de risco de crédito no mercado de derivativos (WESTON *et al.*, 2012). Essas análises envolvem modelos matemáticos complexos e o uso intensivo de computação. Foi empregada uma tecnologia híbrida com a utilização de um *cluster* em que cada nó. Ele era composto por um processador Intel Xeon E5430 conectado a quatro FPGAs Xilinx Virtex-6. Um outro exemplo de aplicação do FPGA em soluções computacionais de alto desempenho é o utilizado pelo grupo de pesquisa HPGC, que busca fornecer soluções para diversas aplicações, com otimizações na aplicação, no algoritmo e na arquitetura. Além de incluir as tradicionais CPU e GPU, o grupo também faz uso do FPGA para propor e implementar suas principais arquiteturas em *High Performance Computing* (HPC). Os principais domínios de aplicação investigados pelo grupo concentram-se em geociências (HPGC, 2015). Porém, mesmo com a integração do FPGA nos *clusters* para prover computação de alto desempenho, há uma quantidade limitada de profissionais que utilizam o FPGA de modo eficiente como um acelerador de hardware e exploram o seu potencial.

Segundo Lawrence *et al.* (2018), a diversidade computacional e a crescente complexidade do software provocaram uma real possibilidade de que os modelos de previsão de tempo e clima cheguem a um abismo que separará o software desenvolvido do hardware disponível. Modelos de previsão do tempo e clima são compostos por "peças" complexas de software que incluem diversos componentes individuais. Cada um desses componentes evolui com a necessidade de melhorar a resolução espaço/tempo do modelo, aumentar a fidelidade em relação aos processos executados e reduzir a incerteza do modelo. Além disso, o tempo de simulação desses modelos é um fator importante, que impacta diretamente na escolha e/ou adaptação do hardware em que ele é executado (LAWRENCE *et al.*, 2018).

O *Brazilian developments on the Regional Atmospheric Modeling System* (BRAMS), por exemplo, é um modelo que ao longo dos anos passou por uma série de alterações e adaptações do software ao hardware disponível no Centro de Previsão do Tempo e Estudos Climáticos (CPTEC) do Instituto Nacional de Pesquisas Espaciais (INPE). Esses avanços são descritos na seção de trabalhos relacionados ao desempenho do BRAMS. Segundo Lawrence *et al.* (2018), o método de melhorias incremental que se ajustam ao ambiente de hardware em mudança é, provavelmente, inadequado para cruzar o abismo entre o hardware disponível e a evolução em termos de software do modelo em um ritmo satisfatório.

A abordagem de aumentar o número de CPU para prover mais capacidade de computação para esses modelos, decorrente de velocidades de clock mais rápidas e de um paralelismo de fácil acesso está no seu final. Por outro lado, prover mais capacidade exigirá a exploração de arquiteturas de computadores mais complexas e de novas rotas para o paralelismo, como por exemplo, modificação/adaptação dos algoritmos existentes e portar tais códigos para plataformas

de computação que possibilitam explorar o paralelismo (LAWRENCE *et al.*, 2018).

Segundo Lawrence *et al.* (2018), essa abordagem “mais do mesmo” irá quebrar à medida que as tendências nas arquiteturas de computadores sugerem que uma abundância de plataformas de computação heterogêneas e complexas será o método de fornecer mais capacidade, como por exemplo a plataforma *Hardware Accelerator Research Program* (HARP)¹. Ela combina CPU e FPGA no mesmo chip, e atualmente foi disponibilizada pela Intel para ser utilizada em um número limitado de projetos de pesquisa em todo o mundo, logo após a compra de uma das duas maiores empresas de FPGA do mundo, a Altera.

Por fim, uma observação importante apontada por Lawrence *et al.* (2018) é a velocidade em que o hardware está evoluindo e não é acompanhada pelo mesmo desenvolvimento dos modelos de previsão de tempo e clima existentes. No caso do BRAMS, por exemplo, a equipe de desenvolvimento do código foi reduzida para dois membros ativos, e o supercomputador Tupã, no qual o BRAMS é executado já encontra-se obsoleto em termos de processamento e manutenção.

1.2 Justificativa

Previsão do tempo e mudanças climáticas estão entre as questões econômicas e sociais mais importantes que a humanidade enfrenta ao longo dos anos, mas também um dos desafios científicos mais complexos. Nas últimas décadas, modelos numéricos tornaram-se um elemento metodológico central para previsão e pesquisa climática. Esses modelos representam os principais aspectos do sistema climático, e a resolução desses modelos exige um poder computacional sempre maior do que o utilizado no presente (PASC, 2014).

Aplicações voltadas para previsão meteorológica possuem estrutura de código fonte bastante complexa (na ordem de 100.000 linhas de código) e são desenvolvidas e utilizadas por grandes comunidades (CUMMING *et al.*, 2013). Contudo, alterações nas arquiteturas de computação de alto desempenho e/ou paradigmas de programação associados a esses códigos sempre foram um desafio.

Há três problemas fundamentais a serem resolvidos (PASC (2014)): 1) esforço para adaptar o código para as novas arquiteturas de hardware, pois este é longo e significativo quando comparado com um ciclo típico de lançamento de novos processadores; 2) otimizações a serem realizadas para um domínio específico (*target*), em que frequentemente incorrem em perdas de desempenho em outras arquiteturas de hardware e; 3) desenvolvimento do código, pois normalmente ele é desenvolvido por cientistas de domínio (meteorologistas), que geralmente, não têm a expertise para otimizar códigos para um alvo específico.

Nesse sentido, a proposta definida durante este doutorado partiu de duas hipóteses:

¹ <https://software.intel.com/en-us/hardware-accelerator-research-program>

- **1ª:** É possível esquematizar transformação de código Fortran para executar em um ambiente de computação heterogênea composto por CPU e FPGA?
- **2ª:** É possível esquematizar transformação de código do BRAMS para um ambiente de computação heterogênea composto por CPU e FPGA com ganho de desempenho?

Dessa forma, essa tese colabora com soluções de otimização para execução de código, por meio de aceleradores específicos em uma plataforma reconfigurável. A solução aqui descrita pode prover adaptabilidade com relação a aplicação alvo e flexibilidade com a inclusão ou retirada de elementos de processamento da arquitetura de hardware.

1.3 Motivação

O BRAMS (*Brazilian developments on the Regional Atmospheric Modeling System*) constituiu-se em uma versão brasileira do RAMS (*Regional Atmospheric Modeling System*) descrito por Pielke *et al.* (1992). O BRAMS foi desenvolvido com os objetivos de criar e aprimorar novos esquemas de parametrizações, incorporar modelos que auxiliam na previsão climática e atualizações para as regiões tropical e subtropical do Brasil (FREIRE; FREITAS; COELHO, 2015).

Atualmente, o BRAMS encontra-se na versão 5.2 e possui aproximadamente 400.000 linhas de código em *Fortran*, sendo esta a versão operacional no CPTEC/INPE do Brasil. O BRAMS tem como principal objetivo fornecer um modelo único para centros regionais de meteorologia brasileiros, com resolução espacial de 5km, configuração padrão utilizada em produção pelo CPTEC/INPE (FREITAS *et al.*, 2017; CPTEC/INPE, 2018).

Para o desenvolvimento deste trabalho de doutorado foram utilizadas 110.000 linhas do total do código do BRAMS, correspondentes ao módulo da dinâmica. Esse código foi disponibilizado por um dos responsáveis pelo código do BRAMS no Brasil, o professor e pesquisador Jairo Panneta. Maiores explicações sobre o código da dinâmica e a disponibilização do mesmo encontra-se no Anexo B.

Alguns dos desafios da área de pesquisa climática são reduzir a incerteza e o tempo computacional. O progresso nesse sentido está, entre outros aspectos, ligado à necessidade de aumentar a gama de escalas tempo-espço, bem como melhorar a representação dos processos físico-químicos nos modelos de tempo e clima. Todavia, ambos os aspectos estão intimamente ligados ao aumento substancial dos recursos de computação exigidos para tais modelos.

Dessa forma, arquiteturas de alto desempenho heterogêneas (CPU, GPU e FPGA) são uma alternativa para atender esses requisitos com alta performance e consumo eficiente de energia. Porém, um ponto negativo é que a adaptação dos códigos para previsão de clima/tempo atuais, de modo a explorar de forma eficaz essas arquiteturas exige um esforço caro e propenso

a erros. Outro fator é o longo ciclo de vida desses códigos, que torna a situação ainda mais crítica se comparado com o ciclo de vida do hardware, que é muito mais curto. Além disso, os modelos atmosféricos são desenvolvidos e utilizados por uma grande variedade de pesquisadores em inúmeras plataformas de computação, o que torna a portabilidade um requisito crucial em qualquer tipo de desenvolvimento (CUMMING *et al.*, 2013).

Nesse sentido, este projeto de doutorado consistiu na implementação e avaliação de um *framework* para coprojeto hardware/software para computação de alto desempenho, que possui como estudo de caso o código do módulo da dinâmica do BRAMS. A arquitetura de hardware proposta no *framework* é inspirada no ambiente de desenvolvimento do Stretch (ARNOLD, 2005), um processador de software configurável para aplicações de computação intensiva. Esse projeto teve como base o dispositivo FPGA como plataforma de desenvolvimento e prototipação da unidade de hardware que compõe o coprojeto hardware/software da dinâmica do BRAMS e a execução desse coprojeto no ambiente de computação heterogênea HARP.

1.4 Objetivos

1.4.1 Geral

Projetar, implementar e avaliar um *framework* para coprojeto de hardware/software da dinâmica do modelo climático BRAMS, usando como plataforma de desenvolvimento e prototipação uma arquitetura heterogênea composta por circuitos programáveis do tipo FPGA e processador Intel Xeon.

1.4.2 Específicos

- Desenvolver técnicas que auxiliem na construção dos blocos de hardware por meio de algoritmos e do uso de linguagem de descrição de hardware de síntese de alto nível;
- Construir uma biblioteca de hardware a ser utilizada para configuração do FPGA como um coprocessador extensível e reconfigurável;
- Prover um ambiente de compilação para geração de código executável para a arquitetura proposta;
- Comparar o desempenho da arquitetura implementada com trabalhos relacionados;
- Analisar o consumo de energia dos blocos de hardware para a arquitetura proposta;
- Disponibilizar uma metodologia de geração de hardware para outros módulos do BRAMS.

1.5 Organização da tese

A tese é composta por 7 capítulos, que estão assim estruturados:

- O Capítulo 2 descreve conceitos sobre coprojetos de hardware/software e o uso do FPGA em HPC.
- O Capítulo 3 reporta os trabalhos relacionados.
- O Capítulo 4 descreve a metodologia empregada para o desenvolvimento do trabalho.
- O Capítulo 5 apresenta o coprojeto hardware/software do modelo climático BRAMS.
- O Capítulo 6 descreve os resultados dos estudos conduzidos para avaliação do *framework*.
- O Capítulo 7 apresenta as contribuições, limitações e oportunidades para trabalhos futuros a partir deste trabalho de doutorado.

FUNDAMENTAÇÃO TEÓRICA

2.1 Considerações iniciais

Neste capítulo são apresentados conceitos de coprojeto hardware/software. Durante o capítulo é explorada a relação entre as características desejadas em um coprojeto, tais como: desempenho, eficiência energética, espaço de coprojeto, mapeamento de aplicação e seleção da plataforma alvo. Também são reportados conceitos sobre FPGA em computação de alto desempenho e tecnologias relacionadas.

O capítulo está organizado da seguinte forma: na Seção 2.2 são apresentados conceitos de coprojeto hardware/software. Na Seção 2.3 é descrito o conceito de *profiling*. Na Seção 2.4 são apresentados fundamentos de computação reconfigurável. Nas Seções 2.5, 2.6, 2.7 são apresentados conceitos sobre o HARP, *OpenCL* e BRAMS, respectivamente.

2.2 Coprojeto hardware/software

A origem do termo coprojeto hardware/software surgiu no início de 1990 para descrever problemas relacionados ao projeto dos Circuitos Integrados (CI). Uma classe de projetistas de microprocessadores de CI passaram a explorar possibilidades de adicionar componentes de hardware padrão em uma placa, porém grande parte do código que eles utilizavam estava na linguagem Assembly. Na década de 90, era claro que sistemas baseados em microprocessador futuramente teriam grande importância nos projetos de CI. Nessa época já estava claro que a lei de Moore possibilitaria criar chips grandes suficientes para incluir CPU e outros subsistemas. Isso levou a duas classes de problemas: metodologias de projeto de sistemas teriam de gerenciar grandes CPUs preprojetadas e o software teria que ser tratado como componente principal no projeto de um chip. Pesquisas resultaram em algumas abordagens para o projeto de software embarcado para CPUs, e esses trabalhos se tornaram as raízes da metodologia de coprojeto

hardware/software (WOLF, 2003).

A atividade de coprojeto consiste no particionamento de uma aplicação em uma parte flexível (software) e uma fixa (hardware). A parte flexível inclui dados de configuração, definições de parâmetros, e outros. A parte fixa é composta por componentes programáveis e reprogramáveis, tais como microprocessadores, coprocessadores, FPGA, dentre outros. Há diversos motivos para a implementação de sistemas baseados em coprojeto, tais como: tamanho do hardware, área ocupada em dispositivos reprogramáveis, consumo energético, aumento de desempenho, flexibilidade e outros (SCHAUMONT, 2010).

Em alguns casos, um coprojeto de hardware/software precisa considerar problemas de otimização para atingir uma determinada métrica de desempenho e, por sua vez, esses só podem ser resolvidos em nível *assembly* (SCHAUMONT, 2010). Por esse motivo, os detalhes de baixo nível da construção do software que irá compor o coprojeto são importantes, afetando diretamente o desempenho do produto final. Para auxiliar na geração do *assembly* a maioria dos compiladores oferecem a funcionalidade de *disassembly* do código objeto.

Decidir entre implementar um projeto totalmente em hardware ou em software é uma decisão difícil e custosa, pois um projeto em hardware possui como fatores mais custosos do que o projeto em software, o esforço de desenvolvimento, tempo gasto, plataforma de prototipação, dentre outros. A implementação em software possui como vantagens a fácil escrita (conjunto de linguagens de programação legadas e amplamente utilizadas), flexibilidade e todo um ferramental disponível e otimizado que inclui rápidos compiladores e grande quantidade de código fonte (SCHAUMONT, 2010; CARDOSO; HÜBNER, 2011; BACON; RABBAH; SHUKLA, 2013).

Contudo, uma implementação em hardware possui uma grande vantagem em relação a uma similar em software, que é o desempenho. Arquiteturas de hardware especializados têm uma maior quantidade de paralelismo do que arquiteturas de software. Nesse caso, o desempenho relativo pode ser medido pela quantidade de trabalho útil realizado por ciclo de *clock*. Dessa forma, implementações altamente paralelas estão em vantagem porque realizam muitas operações ao mesmo tempo (SCHAUMONT, 2010; CARDOSO; HÜBNER, 2011).

No entanto, o desempenho relativo pode não ser um argumento suficiente para motivar a utilização de uma implementação em hardware dedicado, por exemplo, ao comparar um *Application Specific Integrated Circuit* (ASIC) a um processador de propósito geral de última geração. O hardware que compõe o ASIC pode executar várias operações em paralelo, porém o processador funciona a uma frequência de *clock* muito mais alta. Além disso, os processadores modernos são muito eficazes na realização de várias operações por ciclo de *clock* (SCHAUMONT, 2010; GULATI; KHATRI, 2010). Como resultado disso, um programa de software otimizado para a arquitetura desse processador pode superar o projeto de hardware presente em um ASIC. Assim, o desempenho absoluto do software pode ser superior ao desempenho absoluto do hardware. De forma contrastante ao desempenho relativo, o desempenho absoluto considera a frequência do

clock. O desempenho sempre foi um argumento clássico em favor da implementação em hardware dedicado. Contudo, o aumento do desempenho geralmente é obtido reduzindo a flexibilidade da aplicação ou da arquitetura de hardware especializada descrita para a aplicação alvo.

A eficiência energética mede a quantidade de trabalho útil por unidade de energia. Por essa razão, dificilmente será utilizado um processador de estação de trabalho *high-end* em um telefone celular. Quase todos os dispositivos eletrônicos de consumo hoje em dia possuem uma bateria (exemplos iPod, PDA, telefone móvel, etc.). Isso faz com que esses dispositivos apresentem restrições ao consumo de energia, para torná-los utilizáveis e aceitos no mercado. Concomitantemente, esses dispositivos são usados e executam aplicações semelhantes aos computadores pessoais tradicionais de elevado desempenho.

Para que esses dispositivos sejam suficientemente eficientes em relação a energia, eles são projetados e implementados usando uma combinação de software embarcado e componentes de hardware dedicados. Isso é um exemplo bem conhecido e utilizado em coprojeto hardware/software, que consiste na implementação de funções específicas ou custosas do software em hardware fixo, visando eficiência energética, desempenho e outras características (SCHAU-MONT, 2010; GULATI; KHATRI, 2010; CARDOSO; HÜBNER, 2011).

Dessa forma, o desempenho relativo e a eficiência energética são fatores importantes para definir uma implementação em hardware ou em software. No entanto, analisar o “quadro” como um todo, é mais complicado. Na concepção dos sistemas eletrônicos modernos, há muitos *trade-offs* e objetivos conflitantes que vão além do desempenho relativo e da eficiência energética, como por exemplo, complexidade, custo, entre outros.

Uma prática comum é a integração de múltiplos microprocessadores em conjunto com todos os periféricos associados e componentes de hardware em um único chip, os chamados *System-on-Chip* (SoC). No entanto, devido a complexidade dos modernos SoCs a concepção dos mesmos torna-se impraticável sem um planejamento detalhado. Durante a fase de implementação, várias simulações são necessárias e realizadas para testar o projeto inicial, antes de submetê-lo a fase mais onerosa (SCHAU-MONT, 2010). Com relação ao custo, o projeto e fabricação de um *chip* é um processo caro. Em virtude disso, os projetistas de hardware desenvolvem *chips* programáveis, para que possam ser reutilizados em vários produtos ou gerações do produto. O SoC é um bom exemplo desse processo.

O tempo para disponibilizar uma tecnologia no mercado também é um fator importante no projeto de uma arquitetura hardware/software (SCHAU-MONT, 2010). Cada nova geração da tecnologia tende a substituir a mais velha mais rapidamente. Além disso, as novas tecnologias são exponencialmente mais complexas do que a geração anterior. Para a indústria e projetistas isso significa que mais trabalho deve ser concluído no menor espaço de tempo possível, motivando o trabalho em conjunto e de forma simultânea das equipes de hardware e software.

Dessa forma, em um coprojeto hardware/software equilibrar esses fatores e características

possuem significados diferentes a depender da plataforma. Por exemplo, se a plataforma possui um processador de propósito geral, o software é escrito em uma linguagem de programação. Porém, se a plataforma é um FPGA, o “software” é escrito utilizando uma *Hardware Description Languages* (HDL). Quando nesta plataforma está presente um *softcore*, esta também pode executar um software escrito em C, por exemplo. O mesmo se aplica para um *Application-Specific Instruction set Processor* (ASIP), no qual o software pode conter código C, bem como as extensões em hardware do processador.

A Figura 1 ilustra também as diferentes plataformas existentes que podem ser utilizadas em um coprojeto hardware/software. Plataformas de propósito geral, como PPG e FPGA, fornecem suporte a uma ampla gama de aplicações. Enquanto que uma plataforma específica, exemplificada pelo ASIC, é otimizada para executar uma única aplicação.

Entre as plataformas de propósito geral e específico existe a plataforma de domínio específico. A qual é otimizada para suportar um domínio específico de aplicações, como por exemplo, processamento de imagens, processamento de sinais, criptografia, redes, entre outros. Um domínio pode ter subdomínios. Por exemplo, o processamento de sinal é um domínio. Todavia, pode-se distinguir processamento de sinal de voz de sinal de vídeo e plataformas de dispositivo otimizado para cada um deles. Dois exemplos de plataformas de domínio específico são o DSP e o ASIP (SCHAUMONT, 2010).

Um fator importante a ser considerado em um coprojeto hardware/software é o mapeamento da aplicação para uma plataforma, seja ela de domínio específico, de propósito geral ou específico. Cada uma das plataformas ilustradas na Figura 1 apresenta um *trade-off* entre a flexibilidade da aplicação e a eficiência da plataforma. Nesse contexto, a flexibilidade consiste em como uma plataforma pode ser adaptada para diferentes aplicações. Flexibilidade é uma característica desejada, pois é possível alterar a aplicação após a plataforma alvo estar fabricada. Contudo, quando uma plataforma torna-se cada vez mais especializada, a aplicação se aproxima desta, tornando o software mais específico para o hardware (SCHAUMONT, 2010; GULATI; KHATRI, 2010).

A eficiência de uma plataforma é uma característica relativa que pode ser relacionada com o desempenho absoluto ou com a eficiência energética de uma aplicação. Diferentes plataformas podem fornecer níveis distintos de eficiência. No entanto, a escolha de uma plataforma especializada para uma aplicação tende a ser mais eficiente do que uma plataforma geral, pois os componentes de hardware são otimizados para a aplicação (SCHAUMONT, 2010).

Dessa forma, selecionar uma plataforma para uma determinada aplicação, e/ou mapear uma aplicação para uma determinada plataforma é um processo desafiador. No entanto, é possível utilizar uma metodologia de projeto, que consiste em uma sequência de etapas para converter uma especificação em uma implementação. Essas etapas também são utilizadas para identificar e cobrir aspectos durante o mapeamento da aplicação, tais como otimização no uso da memória, desempenho do projeto, uso de recursos, e outras características.

2.3 Técnicas de *profiling*

Técnicas de *profiling* são utilizadas para extrair informações sobre a execução do código fonte de um programa. Essas informações possibilitam localizar partes do código que exigem mais poder computacional. Dessa forma, as informações extraídas pelas técnicas de *profiling* podem ajudar a guiar soluções que visam otimizar o desempenho de aplicações (WANG *et al.*, 2015). As otimizações podem variar de alterações nos algoritmos ou estruturas de dados, projeto de arquiteturas de hardware ou implementação de partes do código em aceleradores de hardware (MISHRA *et al.*, 2012).

O *profiling* de código com o objetivo de reunir informações sobre o fluxo de controle tem recebido atenção ao longo do tempo. No início, as ferramentas possuíam limitações e precisavam ser utilizadas junto com a análise manual (BRAGE; MADSEN, 1994). Com o tempo elas foram melhorando e hoje a maioria das técnicas de *profiling* baseiam-se em análise estática (*off-line*) do programa e com o advento da compilação dinâmica e otimizações na execução, o uso desse tipo de profiles tem aumentado (ARNOLD *et al.*, 2000; ARNOLD; RYDER, 2001; MERTEN *et al.*, 2001; KISTLER; FRANZ, 2001; CHILIMBI; HIRZEL, 2002).

Profilers são ferramenta para *profiling*. Há basicamente dois tipos delas: as que simulam um conjunto de instruções e as que monitoram o programa na arquitetura de hardware. Os profilers baseados em *Instruction Set Simulators* (ISS) são mais detalhados e mais precisos, porém isso incorre em custo da velocidade de execução. Esse tipo de ferramenta é geralmente utilizada por desenvolvedores de arquiteturas de hardware para determinar medidas de custos e vantagens de uma arquitetura. Esse tipo de *profiling* é em média 10 mil vezes mais lento que a execução nativa (LANCASTER, 2011), sendo assim inviáveis para uso com softwares grandes e SoCs, cuja natureza heterogênea constitui grande dificuldade para a concepção de modelos de simulação (TONG; KHALID; MOHAMMED, 2008). Como consequência, as ferramentas que se baseiam na execução do programa no ambiente em que são executadas são as mais conhecidas.

Há várias técnicas de *profiling* que coletam informações do código com base em diferentes granularidades. Os profilers podem inserir marcações no código fonte do programa para obter informações sobre a execução, atividade chamada instrumentação e consiste de três etapas (MILLER *et al.*, 1995): i) ponto de instrumentação, que é o local no código do programa em que a marcação é inserida. Geralmente são pontos na entrada e saída de métodos (*profiling* de arestas) e declarações de chamadas; ii) primitiva de instrumentação, que é a operação para obter uma medida de desempenho. As métricas são baseadas em contadores em hardware ou software e *timers* em hardware; iii) predicado de instrumentação, que são expressões booleanas para controlar a execução de uma primitiva de instrumentação (essencialmente uma declaração *if*).

As marcações podem ser inseridas de forma estática ou dinâmica. Na forma estática, as marcações são inseridas em algum passo do processo de compilação, antes da execução do

programa. Na forma dinâmica, durante a execução do programa a instrumentação é realizada em um processo que gera overhead adicional por agrupar tarefas de *parsing*, *disassembly*, geração de código e decisões em tempo de execução. A instrumentação das duas formas possuem o custo do *overhead* no tempo de execução do programa. Em sistemas embarcados, principalmente os reativos, em que há rigorosas restrições de tempo real, um aumento no *overhead* pode não ser tolerável (SHANKAR, 2010). Como consequência dos problemas oriundos da instrumentação, alguns profilers evitam ou se utilizam da “instrumentação leve” (METZ; LENCEVICIUS; GONZALEZ, 2005) com o objetivo de minimizar ou não provocar *overhead* durante a execução do programa.

Algumas técnicas utilizam um grafo de fluxo de controle para representar o programa como blocos de comandos (BALL; LARUS, 1996). No grafo de fluxo de controle cada bloco representa uma sequência de comandos executados sequencialmente. Um desvio no fluxo de comandos é representado por uma aresta no grafo. As técnicas mais populares incluem informações sobre o fluxo de controle, como blocos, arestas e caminhos. Informações obtidas com *profiling* de blocos podem ser coletadas pelo *profiling* de arestas e estas podem ser coletadas pelo *profiling* de caminhos. Um exemplo é ilustrado na Figura 2.

Figura 2 – Exemplo para *profiling* que não identifica caminhos com maior frequência

Fonte: Ball e Larus (1996).

A tabela representada na figura contém dois diferentes *profiles* de caminho. Ambos induzem as mesmas frequências de execução de arestas mostradas pelas frequências de arestas no grafo de fluxo de controle. No *profile* de caminho, o caminho $Prof_1$ ABCDEF é o mais frequentemente executado. Utilizando a heurística de seguimento de arestas com frequências mais altas o caminho ACDEF é identificado como o mais frequente (BALL; LARUS, 1996), diferindo do *profile* de arestas.

O problema dessas técnicas é o alto custo, em razão do *overhead* decorrente do grande

número de instruções, que exigem manter estruturas de dados para armazenar informações. Além disso, muitas vezes é necessário revisitar várias vezes essas estruturas de dados para obter a granularidade das informações.

No contexto de sistemas embarcados, [Tong, Khalid e Mohammed \(2008\)](#) e [Patel e Rajawat \(2013\)](#) estruturaram as técnicas de *profiling* em três grupos: i) baseadas em software, que podem utilizar simulação ou códigos de monitoramento inseridos nos programas, não dependem de um hardware específico e causam grande overhead; ii) baseadas em hardware, que utilizam recursos como registradores contadores especiais contidos em *Performance Monitor Units* (PMU) com o objetivo de obter medidas de tempo de execução e outros eventos reduzindo o overhead; iii) técnicas que resultaram no desenvolvimento de hardware específicos para *profiling* em FPGA, que interagem com *soft processors* e procuram reduzir o overhead ao mínimo e aumentar a precisão.

[Bala \(1996\)](#) apresenta uma técnica de *profiling* para encontrar caminhos em um ambiente dinâmico. Essa técnica instrumenta cada aresta de um segmento de código com 0 ou 1 e representa cada caminho como um bloco inicial seguido por uma sequência de 0s e 1s. As vantagens dessa técnica são a fácil implementação e com o suporte adequado de um compilador e hardware redução no overhead a quase zero. [Nandy, Gao e Ferrante \(2004\)](#) propõem um *framework* para *profiling*, chamado TFP (*Time-Sensitive, Flow-Specific Profiling*), que extrai padrões sobre o fluxo de controle baseado em informações de tempo a partir do código em execução. [Wang et al. \(2015\)](#) propõem uma técnica de profile baseada em *hot spot* chamada Codem. O fluxo de *profiling* utiliza um cross compilador, um simulador e um profiler. No início do processo, o código fonte é compilado e o arquivo binário executável é carregado pelo simulador em que o programa é executado. Nesse momento, o profiler começa a coletar informações da execução simultaneamente e com isso, as *hot spot* são armazenadas em um *buffer* temporário. No final da execução, as funções *hot* são ordenadas pelo tempo de execução e as frequências são obtidas. Para obter as estatísticas é utilizado um *buffer* de traces preenchido com as instruções executando em cada ciclo de clock.

Há vários métodos de profile e ferramentas desenvolvidas e utilizadas na comunidade acadêmica e comerciais. Como exemplo, ferramentas em nível de código fonte, que dentre elas gprof/gcov é a mais usada com compiladores C para gerar estatísticas sobre o código incluindo tempo de execução e frequências em níveis de função e blocos de código. Gprof possui código aberto.

Técnicas de *profiling* também são aplicadas em nível de código assembly. Como exemplo, SpixTool ([CMELIK, 1993](#)) que analisa comportamentos das instruções com abordagens baseadas em simuladores. Comparando *profiling* a nível de código fonte, informações estatísticas precisas podem ser obtidas nesse nível, mas o grande número de instruções leva a grandes overheads. Atuais técnicas de *profiling* são carentes de soluções integradas com efetivas soluções de desempenho efetivas ([WANG et al., 2015](#)).

2.4 Computação Reconfigurável

Tradicionalmente, a computação foi classificada em computação de propósito geral, realizada por um PPG e aplicações específicas de computação, realizadas por um circuito integrado de aplicação específica os chamados ASICs. De acordo com (BOBDA, 2010) arquiteturas de processamento podem ser classificadas em três grupos principais, considerando o grau de flexibilidade de cada arquitetura. São elas: Processadores de Propósito Geral, Processadores de Domínio Específico e Processadores de Aplicação Específica.

A Figura 3 apresenta a relação entre flexibilidade e desempenho dos tipos de processadores, de acordo com a classificação de Bobda. É possível observar que de um lado estão os processadores de propósito geral e do outro lado estão os processadores de aplicação específica. Entre eles, há processadores que podem oferecer um maior ou menor grau de flexibilidade e desempenho.

Figura 3 – Flexibilidade vs. desempenho de classes de processadores

Fonte: Adaptada de Bobda (2010).

Nesse sentido, a escolha de uma arquitetura em detrimento da outra deve passar inicialmente pela definição da aplicação ou do conjunto de aplicações que se pretende executar. De modo que, uma solução seria a unir a flexibilidade dos processadores de propósito geral com o desempenho de um ASIP na execução das aplicações.

A computação reconfigurável representa um paradigma de computação que derruba uma das barreiras existentes entre o hardware e o software. Separá-los tornou-se cada vez mais indistinto, pois, ambos são programáveis nesse modelo de computação. Além disso, esse

paradigma pode ser visto como um *trade-off* entre a computação de propósito geral e o projeto de circuitos integrados específicos de computação, uma vez que tenta alcançar um equilíbrio entre desempenho, consumo de energia, custo e flexibilidade do projeto.

De modo geral, arquiteturas reconfiguráveis são construídas usando lógica reconfigurável com o objetivo de melhorar o desempenho de aplicações que realizam computação intensiva, visando explorar paralelismo ao projetar e implementar arquiteturas de hardware paralelas reconfiguráveis (HSIUNG; SANTAMBROGIO; HUANG, 2009). Uma das principais motivações para a pesquisa e desenvolvimento de arquiteturas paralelas de computação é o gargalo presente na arquitetura de von Neumann (SEBESTA, 2013). O gargalo refere-se a velocidade de conexão entre a memória e o processador que geralmente limita a velocidade de processamento do computador. Isso ocorre, pois existem situações em que as instruções podem ser executadas mais rapidamente do que sua transferência para o processador executá-las. Uma característica que define uma arquitetura de computação reconfigurável é o tipo de granulação utilizada, que pode ser *fine-grained* ou *coarse-grained*, visto que sistemas diferentes possuem granulações diferentes. Uma arquitetura é dita *fine-grained*, quando manipula dados a nível de *bits*. Já uma arquitetura *coarse-grained* é aquela que manipula grupos de *bits* por meio de complexas unidades funcionais, tais como, Unidades Lógica e Aritmética (ULAs) e Multiplicadores (HAUCK; DEHON, 2008).

De acordo com Bobda (2010), a computação reconfigurável pode ser definida como o estudo da computação envolvendo dispositivos reconfiguráveis. Pois, com a capacidade de reprogramação do hardware presente nesses dispositivos é possível adaptar o hardware para a aplicação alvo.

Por fim, estudos na computação reconfigurável apontam que computadores reconfiguráveis oferecem significativas vantagens sobre microprocessadores de alto desempenho, quando aplicados a um grupo de aplicações. Particularmente aplicações que podem explorar o uso de instruções customizadas e paralelismo maciço em nível de instrução (GOKHALE; GRAHAM, 2005; ZOHOURI, 2018).

2.4.1 FPGA

Um FPGA consiste em um dispositivo programável, composto por três partes principais: um conjunto de blocos lógicos configuráveis, uma matriz programável de interconexões e um conjunto de células de entrada e saída em torno do dispositivo, como ilustrado na Figura 4 (BOBDA, 2010).

Os FPGAs podem ser considerados como um meio-termo entre ASICs e PPGs. A natureza reconfigurável desses dispositivos os tornam mais flexíveis do que os ASICs, e mais eficientes no consumo de energia do que os PPGs. No entanto, esses dispositivos possuem uma programação mais complexa que os ASICs e os PPGs.

Recentes melhorias nas linguagens de descrição de hardware, com *High Level Synthesis*

Figura 4 – Estrutura de um FPGA

Fonte: Bobda (2010).

(HLS), estão possibilitando um maior uso do FPGA em diferentes áreas de aplicação, como por exemplo, em sistemas de computação de alto desempenho. Isso porque tradicionalmente os FPGAs são programados usando HDLs como Verilog e VHDL, que são reconhecidamente mais difíceis de usar e depurar.

Embora, a arquitetura do FPGA seja fixa, ela é composta por blocos de elementos lógicos (*Adaptive Logic Modules (ALM)*), que são organizados na forma de *Loop-Up Tables (LUT)*, registradores e roteamento programável para interconectar os ALMs. Em razão desses elementos, os FPGAs podem ser rapidamente reconfigurados para implementar uma lógica diferente de acordo com a programação especificada pelo projetista de hardware, alterando apenas o conteúdo das LUTs e a configuração de roteamento.

Além dos ALMs, os FPGAs modernos também incluem componentes de lógica fixa ou *hard-logic*, tais como *Digital Signal Processors (DSPs)*, blocos de memória (*Block RAMs*) e diferentes controladores de entrada e saída. Esses componentes implementam lógica especializada que, de outra forma, ocuparia muito espaço se implementada usando LUTs. A Figura 5 ilustra a arquitetura de um FPGA moderno, o Intel Arria 10 (ZOHOURI, 2018).

Figura 5 – Arquitetura do FPGA Intel Arria 10

Fonte: Zohouri (2018).

No entanto, ao longo dos anos, a capacidade dos FPGAs, em termos da quantidade de elementos lógicos presentes, e a velocidade dos mesmos são inversamente proporcionais ao preço e ao consumo de energia, como é ilustrado na Figura 6, que apresenta um estudo para os FPGAs da empresa Xilinx.

2.4.1.1 FPGA em computação de alto desempenho

Supercomputadores reconfiguráveis podem ser descritos como arquiteturas construídas a partir de módulos especializados constituídos por CPUs, FPGAs e memória, combinados por um canal de comunicação de alta velocidade. Os primeiros supercomputadores reconfiguráveis foram o Cray XD1, o SRC-6, e o SGI Altix (HPCL, 2016). Eles são computadores paralelos que assemelham-se a arquiteturas HPC convencional (CPU e GPU), com a adição de chips FPGA. Essas arquiteturas também são conhecidas como *High-Performance Reconfigurable Computing* (HPRC) (EL-GHAZAWI *et al.*, 2008).

Nas arquiteturas HPRC, o FPGA serve como um coprocessador para processadores. Grande parte da aplicação é executada nos processadores, enquanto que o FPGA processa os *kernels* (implementados em hardware) considerados gargalos da aplicação. Do ponto de vista

Figura 6 – Progressão FPGAs da Xilinx em relação capacidade, preço e energia por elemento lógico

Fonte: Ahmad *et al.* (2016).

de desenvolvimento de aplicações, os *kernels* em hardware podem ser construídos utilizando linguagens de descrição de hardware, como VHDL, Verilog, Bluespec e outras. Em alguns casos o fabricante (Maxeler Technologies, por exemplo) disponibiliza ferramentas para conversão de software em hardware ou uma linguagem de descrição de hardware própria (EL-GHAZAWI *et al.*, 2008; WEINHARDT *et al.*, 2015; INC, 2016).

O FPGA é um circuito integrado, projetado para ser reconfigurado de acordo com as necessidades do projetista. Geralmente, ele é programado usando uma linguagem de descrição de hardware. Possui vantagens significativas de desempenho sobre microprocessadores em razão de sua arquitetura altamente paralela e flexível. O paralelismo a nível de hardware permite que aplicações baseadas em FPGA possam operar de 1 a 2 ordens de grandeza mais rápido do que aplicações equivalentes em um processador de propósito geral (VANDERBAUWHEDE; BENKRID, 2013).

Contudo, mesmo com as vantagens citadas, o FPGA não teve uma adoção imediata como plataforma computacional para médias e grandes aplicações. A evolução crescente da tecnologia de silício possibilitou a produção de FPGAs cada vez maiores em termos da quantidade de elementos lógicos presentes no chip e lentamente o FPGA ganhou espaço nas áreas de comunicação e eletrônica automotiva.

No início do ano 2000, o FPGA estava bem estabelecido em quase todas as áreas da computação, exceto a computação de alto desempenho (HPC). Essa exceção pode ser explicada pelo fato de o FPGA mesmo com toda flexibilidade, possuir uma desvantagem quando comparado a um projeto de aritmética customizada em relação a desempenho (VANDERBAUWHEDE; BENKRID, 2013).

Entretanto, por volta de 2003, com mais um avanço na tecnologia de silício, tornou-se impraticável aumentar a frequência dos processadores devido ao reduzido tamanho dos transistores e do superaquecimento gerado. A solução para esse problema foi o surgimento das arquiteturas multicore.

As arquiteturas multicore rapidamente passaram a ser adotadas para o uso de HPC. Porém, na tentativa de alcançar maior desempenho na execução de uma aplicação, não bastava apenas aumentar a quantidade de processadores. Os modelos de programação desenvolvidos até então eram orientados ao processamento sequencial e não paralelo (VANDERBAUWHEDE; BENKRID, 2013). Desse modo, a ideia de usar FPGAs em HPC ganhou cada vez mais força e utilizadores. Como exemplo, a Intel, disponibilizou em nível de pesquisa e experimental chips contendo um processador Xeon Phi e um FPGA, ambos integrados no mesmo chip. Isto certamente não era previsto há uma década atrás para o uso do FPGA (VANDERBAUWHEDE; BENKRID, 2013). Já é possível hoje em dia, para uma solução do usuário, adquirir uma placa FPGA e integrá-la em uma máquina por meio de um barramento PCI Express.

Um fator vantajoso da utilização do FPGA em HPC é que mesmo com a arquitetura multicore e a introdução de novos paradigmas de programação, ainda existe uma grande dificuldade em alcançar velocidades escaláveis nesse modelo de computação. Outro fator é que com a diminuição do transistor é possível produzir FPGAs cada vez maiores. O FPGA por operar em frequências mais baixas ainda não atingiu a densidade de potência limite da CPU (VANDERBAUWHEDE; BENKRID, 2013).

A flexibilidade do FPGA permite realizar várias configurações de hardware e a sua natureza paralela possibilita a criação de aceleradores para atender as necessidades da aplicação alvo. Além disso, esta flexibilidade fornece a oportunidade de criar máquinas ideais para grandes aplicações, ao contrário de uma década atrás, em que devido ao tamanho (quantidade de elementos lógicos) reduzido do FPGA, essas aplicações seriam grandes demais para serem mapeadas para o mesmo.

É possível observar uma grande utilização do FPGA, seja para emulação ou como plataforma final da aplicação. Ademais, as emulações realizadas correspondem a hardwares customizados para uma aplicação específica, fazendo uso da flexibilidade do FPGA para projetar formas e/ou técnicas de computação não disponíveis em processadores convencionais. Dessa forma, com o uso do FPGA os projetistas emulam o futuro da computação de alto desempenho, a qual não estará restrita apenas ao uso de PPGs, arquiteturas multicore e GPUs.

Uma série de tecnologias para HPC foram recentemente lançadas e/ou melhoradas. A geração de processadores Xeon Phi coprocessador da Intel, um híbrido entre um processador de propósito geral e um acelerador. Esta classe de processadores teve um grande impacto na indústria com o seu *design* inovador que combina uma arquitetura de múltiplos núcleos com a produtividade de propósito geral.

2.5 HARP

O HARP (*Heterogeneous Architecture Research Platform*) é um programa criado pela Intel, que tem com um dos seus objetivos fomentar a pesquisa em FPGA, com o intuito de criar soluções tecnológicas que envolvam o futuro dos processadores da Intel, como por exemplo, o processador Intel Xeon E5 v4 com FPGA Arria 10 integrado. Outro objetivo é fomentar a pesquisa de sistemas de computação que focam em novas abordagens de integração de CPUs com aceleradores no nível de software e hardware (GUPTA, 2015; INTEL, 2018a).

O hardware reconfigurável disponível no HARP conecta um processador ao FPGA por meio de uma conexão do tipo *QuickPath Interconnect* (QPI) de alta velocidade, que melhora a largura de banda do sistema. Nessa arquitetura, o FPGA pode ser programado para funcionar como um acelerador para a aplicação alvo, podendo prover desempenho e eficiência no consumo de energia. Outro objetivo desse programa ao fornecer acesso antecipado a esse processador a pesquisadores de diferentes países e projetos foi gerar uma comunidade de pesquisa de classe mundial, tendo como objeto de estudo essa nova arquitetura de processamento e liderar por parte da Intel esta tecnologia a ser disponibilizada no futuro para os *data centers* (GUPTA, 2015).

A infraestrutura do HARP dispõe de processadores com a tecnologia Intel Broadwell e o FPGA Arria 10 integrados. Foram disponibilizados 2 clusters, cada um com 10 nós. Um cluster está localizado no *Texas Advanced Computing Center*, na Universidade do Texas em Austin, Estados Unidos, e o outro em *Paderborn Center for Parallel Computing*, na Universidade de Paderborn, Alemanha. O programa conta com a participação de mais de 150 universidades ao redor do mundo (GUPTA, 2015).

Segundo Metz (2015), embora o mundo esteja caminhando para algo como FPGAs, os *chips* tradicionais utilizados em servidores continuarão a ser usados. O HARP foi idealizado para explorar esse novo conceito que une em um mesmo *chip* uma CPU com um FPGA.

Arquiteturas multicore usam coerência de cache, com memória de baixa latência e de alta velocidade para prover comunicação entre os *cores* do processador. No HARP a interface do tipo QPI é usada para integrar os aceleradores reconfiguráveis programados no FPGA com os múltiplos núcleos do processador. Dessa forma, todos os dispositivos compartilham a memória do sistema, intermediados por protocolos de coerência de cache.

A primeira versão do HARP foi disponibilizada em 2015 para a comunidade acadêmica. A plataforma do sistema de processamento disponibilizada é ilustrada na Figura 7. No HARP 1 o processador é um Intel Xeon E5-2680 v2 e o FPGA é um Altera Stratix V. A motivação apresentada por Gupta (2015) para a utilização dessa arquitetura é a de aprimorar o desempenho dos processadores em *data centers* usando aceleradores em FPGA.

Em meados de 2017 o HARP 2, segunda versão do programa, foi disponibilizado para a comunidade de pesquisa. A plataforma de processamento do HARP 2 é ilustrada na Figura 8. Por fim, no início de 2018, a Intel anunciou o primeiro processador comercial com um FPGA

Figura 7 – Plataforma processador IvyTown Xeon + FPGA Stratix V

Fonte: Gupta (2015).

integrado, a Figura 9 ilustra o chip do processador Intel Xeon Scalable integrado com um FPGA Arria 10.

De acordo com Huffstetler (2018), o lançamento do processador Intel Xeon Scalable é um importante resultado após a aquisição da Altera pela Intel. O processador inclui um FPGA Arria 10 GX 1150, que fornece até 160 Gbps de largura de banda nas interfaces de entrada e saída e uma interface de coerência de cache fortemente acoplada. O FPGA possui sua própria memória cache e faz o uso de memória compartilhada com o processador por meio da interface de coerência de cache ligada ao barramento Intel *Ultra Path Interconnect* (UPI).

O Intel UPI permite acesso contínuo aos dados, independentemente de onde os dados residam (cache do processador, cache do FPGA ou na memória), sem a necessidade de armazenamento redundante de dados e transferências de acesso direto à memória via *Direct Memory Access* (DMA). A coerência de dados também reduz a complexidade da programação da aplicação e economiza ciclos de CPU, que seriam desperdiçados para determinar quais dados estão mais atualizados.

Segundo Huffstetler (2018), os FPGAs continuam como uma parte importante do portfólio de soluções otimizadas para cargas de trabalho em *data centers* provida pela Intel. O futuro da tecnologia depende também de contínua melhoria nas ferramentas disponíveis para facilitar o uso dos FPGAs da Intel e de outros aceleradores principalmente em *data centers*.

O projeto de arquitetura do HARP possui algumas semelhanças e contrastes com outros sistemas modernos de computação heterogêneo do tipo CPU-GPU. Nos sistemas CPU-GPU, antes de iniciar o processamento dos dados na GPU é necessário primeiro transferir os dados para a memória do dispositivo. Por outro lado, o sistema HARP baseia-se em um protocolo de

Figura 8 – Plataforma integrada Intel Xeon + FPGA Arria 10

Fonte: Gupta (2016).

coerência de cache, com o objetivo de facilitar o compartilhamento de dados entre a CPU (host) e o FPGA (GUPTA, 2016; CALDEIRA JERONIMO C. PENHA; PEREIRA, 2018). O Acelerador programado no FPGA solicita os dados em tempo de execução às APIs de hardware/software disponibilizadas pela Intel. Desse modo, em contraste com o sistema GPU não há interferência explícita do desenvolvedor na transferência dos dados a serem processados pelo acelerador.

Embora as APIs da Intel criem uma camada de abstração para o desenvolvedor, eliminando a necessidade de criar mecanismos de baixo nível para acesso aos dados entre a CPU e o FPGA, algumas limitações dificultam o uso dessas ferramentas. Por exemplo, projetar e codificar um acelerador exige experiência em FPGAs, e mesmo com a utilização de linguagens de alto nível para descrever o hardware, como Intel *OpenCL*, há restrições na programação para extrair um efetivo paralelismo. Outro ponto é conhecer os detalhes de otimizações do compilador para FPGA no processo de síntese do hardware, algo que é recomendado em documentos disponibilizados pela Intel (CALDEIRA JERONIMO C. PENHA; PEREIRA, 2018).

Figura 9 – Intel Xeon Scalable integrado com um FPGA Arria 10

Fonte: [Huffstetler \(2018\)](#).

Maiores detalhes sobre este programa e arquitetura do processador são confidenciais e não podem ser divulgados devido aos acordos de sigilo da informação (*Non-Disclosure-Agreements*) assinados por membros do Laboratório de Computação Reconfigurável do ICMC/USP. Maiores detalhes ainda não foram divulgados em sua totalidade pela própria Intel.

2.6 OpenCL

OpenCL (Open Computing Language) é um padrão aberto (*open-source*) de programação para ambientes computacionais heterogêneos que usa um modelo host/dispositivo. Uma aplicação em *OpenCL* é dividida em duas partes: 1) o host, que é executado pela CPU e pode ser escrito em qualquer linguagem de programação, desde que exista um compilador compatível com as rotinas do *OpenCL* e 2) o código do kernel. O *OpenCL* fornece uma *Application Programming Interface* (API) de programação necessária para controlar o acelerador e prover a comunicação entre o host e o acelerador ([KHRONOS, 2018](#)). A Figura 10 ilustra o modelo de programação *OpenCL*.

No fluxo típico de operação *OpenCL*, inicialmente todos os dados necessários são alocados na memória do host. Então, esses dados são transferidos para a memória do dispositivo, usando as respectivas funções *OpenCL*. Em seguida, o kernel é carregado e executado no dispositivo, em que as entradas são lidas e as saídas são gravadas na memória do dispositivo.

Figura 10 – Modelo de programação *OpenCL*

Fonte: Gupta (2016).

Após o processamento, os dados de saída são transferidos da memória do dispositivo para o host.

Com um número cada vez mais crescente de sistemas de computação heterogêneos que incluem CPUs, GPUs, FPGA e outros tipos de processadores altamente paralelos, possibilitar que os desenvolvedores de software aproveitem ao máximo essas plataformas de processamento heterogêneo é de grande importância para a evolução contínua do desenvolvimento tanto do software quanto do hardware (KHROS, 2018). O processamento paralelo é uma realidade em praticamente todos os segmentos que fazem uso de sistemas computacionais. Arquiteturas de processadores modernos adotaram o paralelismo como um caminho importante para o aumento do desempenho. Uma consequência da evolução foi que quanto mais se aumentou a frequência do processador o ganho de processamento estagnou. Como alternativa para contornar essa barreira foram adicionados múltiplos núcleos de processamento nesses processadores.

Nesse sentido, desenvolver sistemas para plataformas heterogêneas de processamento paralelo é um desafio, já que as abordagens tradicionais de programação para CPUs e GPUs de vários núcleos são muito diferentes e essa diferença cresce quando adicionado o FPGA. Com foco no paralelismo, a programação em *OpenCL* baseia-se na escrita de funções que podem ser executadas em múltiplas instâncias simultâneas (KHROS, 2018; INTEL, 2018b).

Em *OpenCL* é possível especificar dois modelos de programação para um mesmo algoritmo, são eles:

- **Paralelismo de dados (*NDRange*):** são criados múltiplas instâncias de trabalho para a execução do kernel, cada uma chamada *Work-Item* (WI). Os WIs são agrupados para formar um *Work-Group* (WG). De modo que, para executar uma aplicação, os WIs são distribuídos em vários WGs. Para sincronizar os WIs dentro de um WG é usado o mecanismo de barreira, que possibilita que os dados possam ser compartilhados entre os WIs usando a memória local, que é de rápido acesso (*on-chip*). No entanto, a única

maneira de compartilhar dados entre diferentes WGs é através da memória global que possui acesso lento por ser *off-chip*. O Código-fonte 1 é um exemplo de kernel usando o modelo NDRange.

- **Paralelismo de tarefas (*Task*):** um único item de trabalho é instanciado para a execução do kernel. Nesse modelo é criado um *pipeline* no qual os dados são enfileirados e executados. Nesse modelo não há replicação do hardware, diferente do modelo *NDRange*. A paralelização em nível de iteração em forma de vetorização pode ser obtida nesse modelo de programação por desenrolamento de *loop* (*loop unrolling*). O Código-fonte 2 é um exemplo de kernel usando o modelo *Task*.

Código-fonte 1 – Kernel no modelo de programação NDRange

```
1:  __kernel void m_ndrange(__global float* x, __global float* y)
2:  {
3:      int i = get_global_id(0);
4:      x[i] = y[i];
5:  }
6:
```

Código-fonte 2 – Kernel no modelo de programação *Task*

```
1:  __kernel void m_task(__global float* x, __global float* y, int
    size)
2:  {
3:      for (int i = 0; i < size; i++){
4:          x[i] = y[i];
5:      }
6:  }
7:
```

2.6.1 Intel OpenCL SDK

O Intel *OpenCL Software Development Kit* (SDK) para FPGA fornece as APIs necessárias para programar e executar aplicações que usam aceleradores em FPGAs da Intel conectados via PCI-E ou SoC, de modo semelhante a uma GPU e outros aceleradores (INTEL, 2018b; ZOHOURI, 2018).

Todo ferramental necessário para comunicação entre o FPGA e a memória off-chip, bem como os drivers necessários para conexão via PCI-E e DMA entre o host e o FPGA são fornecidos pelo fabricante da placa ou pela própria Intel, o chamado *Board Support Package*

(BSP). O BSP retira do programador a necessidade de fornecer ou configurar manualmente as interfaces de comunicação entre o FPGA e o host, como é feito nos projetos tradicionais de FPGA baseados em HDL.

Alguns BSPs também fornecem a possibilidade de enviar e receber dados usando interfaces de rede integradas ao FPGA. Ao contrário de CPUs e GPUs, a compilação em tempo de execução de *kernels OpenCL* não é possível para FPGAs devido ao tempo muito longo de compilação. O kernel *OpenCL* precisa ser compilado offline, em seguida, carregado em tempo de execução pelo código do host para reprogramar o FPGA e executar o aplicativo (INTEL, 2018b).

A Figura 11 ilustra o fluxo completo de execução de uma aplicação que faz uso em conjunto de uma CPU e um dispositivo FPGA usando Intel *OpenCL* SDK, desde a especificação do arquivo *OpenCL* (.cl), passando pelo compilador AOC e a geração do arquivo binário a ser programado no FPGA (.aocx).

As GPUs normalmente consistem em um conjunto de unidades de processamento de granularidade grossa (*coarse-grained*), em que cada unidade contém um número fixo de processadores que trabalham com granularidade fina (*fine-grained*), uma quantidade fixa de blocos de memória do tipo *scratchpad* e *cache* L1. No modelo de programação NDRange, um WG geralmente é mapeado para uma das unidades de granularidade grossa e cada WI é mapeado para um dos processadores de granularidade fina (ZOHOURI, 2018).

No entanto, a decomposição de processamento que ocorre na GPU em processadores para granularidade grossa e fina não existe em FPGAs. Por padrão, usar o modelo de programação NDRange em FPGAs não resultará em paralelismo em nível *thread* e, em vez disso, o compilador gerará uma unidade de computação implementada como um *pipeline*, com todos os WIs e WGs sendo executados nesse *pipeline*. Cada região entre duas barreiras em um kernel NDRange será mapeada para um *pipeline* separado, com cada *pipeline* sendo liberado após todos os WIs alcançarem a barreira (ZOHOURI, 2018).

O compilador *OpenCL* também é responsável por criar e executar de forma automática o *pipeline* de execução para os *work-groups*, de modo a permitir que vários *work-groups* estejam em execução simultânea na mesma unidade de computação para maximizar a eficiência de todos os *pipelines* na unidade de computação. Isso é feito a um alto custo usando mais blocos de memória RAM do FPGA (INTEL, 2018b; ZOHOURI, 2018).

A Figura 12 ilustra os *pipelines* gerados nos modelos de programação NDRange e Task do Intel *OpenCL*.

2.7 BRAMS

O BRAMS é um modelo numérico de previsão meteorológica projetado para simular circulações atmosféricas, que vão desde ondas de escala planetária até grandes massas turbulentas

Figura 11 – Fluxo de programação Intel *OpenCL* para FPGA's da Intel

Fonte: [Intel \(2018b\)](#).

no limite da camada planetária.

O BRAMS teve a sua base de desenvolvimento no RAMS, contudo, ao longo dos anos,

Figura 12 – Pipelines para os modelos de programação *Task* e *NDRange*

Fonte: Elaborada pelo autor.

ele desenvolveu a sua própria identidade e divergiu do RAMS adicionando novos recursos e modificações que tiveram impacto direto na representação numérica de processos físicos fundamentais em regiões tropicais e subtropicais, como é o caso do Brasil (FREITAS *et al.*, 2005; FREITAS *et al.*, 2009).

Além disso, o BRAMS acopla um modelo fotoquímico, um módulo completo para processos químicos na atmosfera e aerossóis, bem como um modelo de última geração para simular energia, água, carbono e outros ciclos biogeoquímicos. O acoplamento desses modelos ampliam as funcionalidades originais do RAMS, tornando o BRAMS um sistema de previsão de tempo e clima que integra de forma completa um modelo de previsão e estudos ambientais (FREITAS *et al.*, 2017).

O BRAMS é um dos modelos utilizados operacionalmente no CPTEC e em vários outros centros regionais de previsão do tempo no Brasil. Previsões simultâneas de clima e composição atmosférica estão disponíveis em tempo real, incluindo fumaça de incêndios de vegetação. Desde janeiro de 2013, o BRAMS é executado no supercomputador Tupã do CPTEC, utilizando 9600 *cores* de processamento. Onde são realizadas as previsões meteorológicas regionais duas vezes ao dia em uma grade de 5 km e sobre um domínio que cobre toda a América do Sul e os oceanos vizinhos (FREITAS *et al.*, 2017). Além de estar operacionalmente no CPTEC, o BRAMS também tem sido objeto de estudo, sobretudo na área de estudos numéricos, com pesquisas em várias universidades dentro e fora do Brasil, centros de pesquisa que abordam tempestades locais, ilhas de calor urbano, poluição urbana e remota, interações aerossol-nuvem-radiação, ciclos de

carbono e água sobre a Amazônia, cinzas vulcânicas dispersão, etc. (FREITAS *et al.*, 2017).

De modo geral, o BRAMS é composto por três grandes componentes, são eles:

- Modelo Atmosférico

Esse modelo é representado pela resolução das equações não hidrostáticas compressíveis descritas por Tripoli e Cotton (1982), e é equipado com um esquema de agrupamento de múltiplas redes que permitem que as equações do modelo sejam resolvidas em qualquer número de malhas computacionais de resolução espacial. Além disso, possui um conjunto de parametrizações físicas adequadas para simular processos como trocas de ar na superfície, turbulência, convecção, radiação e microfísica de nuvens (CPTEC/INPE, 2018).

- CATT-BRAMS

O CCATT (CATT-BRAMS (*Coupled Aerosol-Tracer Transport model to the Brazilian developments on the Regional Atmospheric Modeling System*)) é um modelo de transporte euleriano acoplado ao BRAMS para simular os processos de transporte, dispersão, transformação química e remoção associados a gases e aerossóis. O CCATT modela o transporte com a simulação do estado atmosférico pelo BRAMS, usando as mesmas parametrizações dinâmicas e físicas (CPTEC/INPE, 2018).

- JULES

Os processos que ocorrem na superfície da terra e ciclo do carbono são simulados utilizando o JULES (*Joint UK Land Environment Simulator*). A integração desse simulador ao BRAMS resultou em um novo sistema que representa o estado da arte na modelagem atmosférica regional, incluindo processos químicos totalmente acoplados, aerossóis e ciclos biogeoquímicos (CPTEC/INPE, 2018). Por fim, o novo sistema aumenta significativamente a capacidade de simular processos envolvendo interações ar-superfície devido à capacidade do JULES de simular fotossíntese, respiração e vegetação dinâmica, entre outros processos (CPTEC/INPE, 2018).

Para cada iteração (*timestep*) do modelo BRAMS, o componente atmosférico fornece ao JULES a velocidade atual do vento próximo da superfície, temperatura do ar, pressão, água condensada e fluxos de radiação descendente, bem como vapor de água, dentre outros. Após seu processamento, o JULES avança suas variáveis de estado ao longo do tempo e o resultado do seu processamento alimenta o componente atmosférico (CPTEC/INPE, 2018).

O componente químico (CATT) realiza emissões não biogênicas, deposição, transporte, reatividade química e processos de aerossol, incluindo fornecer informações de processamento para a meteorologia, fechando o ciclo (CPTEC/INPE, 2018). A Figura 13 ilustra alguns dos processos simulados pelos três componentes no modelo BRAMS.

Figura 13 – Interações entre os componentes do BRAMS

Fonte: CPTEC/INPE (2018).

2.7.1 Dinâmica do BRAMS

A adequada representação dos fenômenos físicos em modelos numéricos da atmosfera é imprescindível para obter previsões confiáveis. Na primeira camada da atmosfera, que está em contato direto com o solo e com os oceanos, um dos efeitos que não podem ser negligenciados é o da turbulência, devido ao transporte das massas de ar que ocorrem sob esse efeito por difusão (FREITAS *et al.*, 2017).

A dinâmica é o principal componente atmosférico de um modelo numérico de previsão de tempo. Ela é responsável pela movimentação do fluido (atmosfera) de acordo com forças, tais como força de Coriolis, aquecimento de superfície por radiação e estímulos vindos da própria atmosfera. Esse conjunto de características torna a dinâmica um componente relevante e essencial para a previsão do tempo (FREITAS *et al.*, 2017).

O BRAMS é um modelo numérico de previsão do tempo altamente customizável em termos de módulos que podem ser executados em conjunto (por meio do arquivo de configuração RAMSIN). Porém para a dinâmica representar menos importância no modelo, basta colocar o módulo da química (CATT-BRAMS) para executar junto com a dinâmica, tal fato pode ser observado com o profile ilustrado na Figura 14 (CPTEC/INPE, 2018).

De acordo com o profile, é possível observar que o módulo da química corresponde a 47.76% da execução, enquanto que a dinâmica corresponde a 30.89%, conforme é ilustrado na

Figura 14. Contudo, é importante destacar que sem a dinâmica não há previsão do tempo. A química pode não ser executada para que haja previsão do tempo, mas a dinâmica não tem essa opção no modelo (CPTEC/INPE, 2018).

TRABALHOS RELACIONADOS

3.1 Considerações iniciais

Neste capítulo é apresentado o estado da arte sobre coprojetos hardware/software. Foram conduzidas revisões de literatura ao longo do doutorado cujos resultados são aqui apresentados.

Este capítulo está organizado da maneira a seguir. Na Seção 3.2 são apresentados os trabalhos relacionados a coprojetos hardware/software. Na Seção 3.3 são descritos trabalhos relacionados ao BRAMS.

3.2 Trabalhos relacionados a coprojetos hardware/software

Um dos primeiros esforços em coprojetos foi o sistema SOS de [Prakash e Parker \(1992\)](#) da Universidade do Sul da Califórnia, que poderia sintetizar uma topologia de multiprocessadores e escalonar e alocar processos no multiprocessador. O sistema formulava a síntese do problema como uma mistura de programação linear com inteiros, de forma que o sistema era lento e não conseguia lidar com grandes problemas, apesar dessas limitações esse estudo foi um trabalho inicial importante ([WOLF, 2003](#)).

A partir daí, o particionamento hardware/software surgiu como um importante passo na criação de modelos de algoritmos. Dois sistemas antigos, Vulcan ([GUPTA; MICHELI, 1993](#)) da Stanford e Cosyma ([ERNST; HENKEL; BENNER, 1993](#)) da Universidade Técnica de Braunschweig, despontaram como abordagens complementares a estes problemas. A Figura 15 ilustra como o particionamento hardware/software mapeava um projeto na arquitetura alvo. Um sistema inclui uma CPU e um ou mais CIs específicos da aplicação conectados por um barramento. Uma particularidade é que nos primeiros projetos, o ASIC se comportava como um acelerador e não como um coprocessador. Em razão disso, o sistema alocava muitas tarefas computacionalmente intensivas para o ASIC, enquanto tarefas mais simples eram direcionadas

do hardware para a CPU ([WOLF, 2003](#)).

Figura 15 – Arquitetura alvo para particionamento hardware/software.

Fonte: [Wolf \(2003\)](#).

Um ponto a se destacar é que desde os primeiros FPGAs o interesse por essa tecnologia aumentou em razão da complexidade relativamente alta e da facilidade de programar um circuito e reprogramá-lo caso ele não atendesse aos requisitos para os quais foi proposto. Basicamente ocorreram dois problemas para a adoção dos FPGAs: i) o mapeamento da tecnologia, no qual transformar uma descrição de circuito utilizando portas lógicas padrão e elementos RTL em uma descrição usando os elementos de um FPGA; ii) implementar grandes circuitos em FPGAs, pois apesar das novas gerações de FPGAs serem mais velozes, complexas e com um número maior de portas lógicas em um chip, circuitos com várias dezenas ou centenas de milhares de portas ainda era um desafio.

Soluções tentando resolver esses problemas foram surgindo, como [Wendling e Rosentiel \(1994\)](#) que definiu um ambiente de hardware com suporte a implementações de grandes circuitos usando um conjunto de FPGAs. [Schmit e Thomas \(1995\)](#) compararam duas aplicações em software e hardware baseado em FPGA. Uma aplicação era baseada em modelagem de Markov (HMM) e a outra era um controlador Fuzzy. Ambas possuíam paralelismo significativo e consistiam principalmente em operações de adição e lógica utilizando operandos em ponto fixo. Diferentes algoritmos foram utilizados para as comparações e os resultados diferiram de acordo com eles. Os autores enfatizaram a dificuldade em fazer conclusões gerais sobre a utilidade de computação baseada em FPGA para esses domínios de aplicação, e concluíram que fatores externos determinam quando computação baseada em FPGA é viável. Como consequência das diferentes possíveis implementações de hardware e software é difícil ou impossível migrar implementações em software diretamente para implementações baseadas em FPGAs.

[Abbott, Athanas e Tarmaster \(1995\)](#) descreve a implementação de um filtro no SPLASH-2, que é um processador reconfigurável, para ilustrar a eficácia do paradigma de computação personalizada. Essa abordagem explora problemas de processamento de imagens como computação repetitiva e altas taxas de transferência de dados. Os objetivos dessas iniciativas anteriores

baseiam-se em prototipagem, porém nessa época começou a aumentar o interesse em usar FPGAs como sistemas para coprojetos hardware-software. Gao e Rees (1996) compara arquiteturas existentes e propõe o projeto de um *ASynchronous Array Processor* (ASAP) configurável e o desenvolvimento de um modelo de computação para o processador. (KIMURA *et al.*, 1997) mostra um método para coprojetos hardware/software para um sistema com um coprocessador reconfigurável a partir de FPGAs.

Uma abordagem para particionamento hardware/software usando programação de inteiros (IP) é apresentada em Niemann e Marwedel (1997). As vantagens de usar IP é que resultados ótimos são calculados para uma função objetivo escolhida. A abordagem de particionamento funciona de forma automática e apoia sistemas multiprocessadores, interfaces e compartilhamento de hardware. Contrastando com outras abordagens que adotam estimativas especiais, os autores usam compilação e ferramentas de sínteses para estimativas de custo. O tempo maior para calcular o custo é compensado por melhorias na qualidade dos valores e, assim, poucos passos de iteração são necessários para o particionamento.

Rapidamente cresceu o número de pesquisas em coprojetos hardware/software. Nos anos seguintes os pesquisadores abordaram vários problemas e os trabalhos que foram desenvolvidos possibilitaram aplicar o coprojetos a uma gama mais ampla de sistemas Wolf (2003). Gschwind, Salapura e Maurer (2001) descreve o uso de um processador reconfigurável baseado na arquitetura *Reduced Instruction Set Computer* (RISC) como ponto de início para o projeto de um processador específico da aplicação.

Técnicas de coprojetos hardware/software vem sendo utilizadas com o objetivo de prototipagem para acelerar cálculos ou partes de código, aplicações de controle climático, processamento de imagens, metodologia de projeto de computação reconfigurável, ambiente de teste, algoritmos de reconstrução de árvores filogenéticas, informações de profiling, sistemas de controle fuzzy configuráveis, robótica móvel, sistemas neuro-fuzzy, aplicações de controle embarcadas, kernel multithreaded RTOS, sistemas de comunicação, método de cosimulação, processamento de sinais, redes neurais, sistemas MIMO (*Multiple-input-multiple-output*), CI de controle de movimento, processamento de dados hiperespectrais, sistema de tempo real de tomografia, dentre uma infinidade de aplicações que exploram as vantagens de uma arquitetura de coprojetos hardware/software (GONZALEZ *et al.*, 2015; GONZÁLEZ *et al.*, 2016; GONZALEZ *et al.*, 2018; FIRDAUS; MERIBOUT, 2016).

Existem diversos trabalhos relacionados ao coprojetos de hardware/software, e também alguns *frameworks*. Contudo, são poucas as soluções que propõem a integração de uma CPU de propósito geral com um FPGA para computação de alto desempenho em um coprojetos de hardware/software utilizando o FPGA como um coprocessador ou como uma unidade de processamento extensível da CPU convencional para acelerar partes da aplicação alvo. Geralmente, as soluções para esta abordagem se aproximam ou são ferramentas comerciais em que os arquivos fonte não estão disponíveis, como por exemplo, o processador Stretch (STRETCH, 2018).

GPUs e FPGAs são duas plataformas que apresentam vantagens para computação de alto desempenho em conjunto com processadores de propósito geral. A combinação CPU-GPU já é bastante utilizada e vem sendo consolidada. Esta consolidação foi alcançada com a programação em linguagens de alto nível, como C/C++ com suporte a um rico conjunto de ferramentas que possibilita a geração de *profiles* e *debug* do código e ao aumento de desempenho na computação. Por sua vez, a descrição do paralelismo a ser executado na GPU ocorre por meio de linguagens de programação específicas, representadas por *Compute Unified Device Architecture* (CUDA) e *OpenCL*.

O trabalho descrito em [Thoma et al. \(2015\)](#) apresenta um *framework open source* que provê a comunicação FPGA-GPU via barramento PCIe. Para iniciar a comunicação e transferência de dados entre os dispositivos é utilizada uma CPU presente na arquitetura. Após o setup inicial, o FPGA realiza o controle da comunicação usando a técnica de DMA. O foco principal do estudo descrito é alcançar altas taxas de comunicação entre as duas plataformas. Para os experimentos foram utilizados dois FPGAs da Xilinx (Virtex-5 e Virtex-7). Na Virtex-5 a taxa de comunicação alcançada foi 200 MB/s com a configuração do PCIe geração 1.1. Já na Virtex-7 a taxa foi 2.4 GB/s com a configuração do PCIe geração 2.1. Nos testes foram utilizadas aplicações que consistem em operações DMA de leitura e escrita variando o tamanho dos dados de 32 *bytes* até 8 MB. Como resultado, observou-se que a operação de escrita foi mais eficiente que a de leitura de dados entre FPGA-GPU. A Figura 16 ilustra a arquitetura de um acelerador contendo um PC padrão com placas PCIe.

Figura 16 – Arquitetura de um acelerador: processador de propósito geral conectado a uma GPU e FPGA

Fonte: [Thoma et al. \(2015\)](#).

Sabe-se que o poder computacional fornecido por CPUs convencionais para computação de alto desempenho não é suficiente. Do mesmo modo, o uso de ASICs ou supercomputadores é uma solução cara e em alguns casos sua adoção é impraticável. Nesse contexto, placas de FPGA, a preços razoáveis, são uma alternativa e podem ser utilizadas para atualizar PCs padrão ofertando flexibilidade e poder computacional. Isso é possível com o mapeamento de *kernels*

da aplicação alvo para o FPGA, o que possibilita a construção de hardware especializado, de forma a alcançar um maior desempenho a um custo mais baixo em relação a aquisição de um supercomputador (WEINHARDT *et al.*, 2015).

Por outro lado, placas de FPGA ainda não são amplamente utilizadas para acelerar aplicações em um PC padrão, nem em HPC. Isso ocorre pois o projeto e implementação de hardware é mais difícil e demorado do que o desenvolvimento de software. Além disso, grande parte dos aceleradores de hardware são soluções otimizadas de forma *ad hoc* para uma placa de FPGA específica. O uso de outras placas FPGA requer portabilidade, o que normalmente é um processo demorado e caro ou reescrita completa do projeto (BOBDA, 2007; BACON; RABBAH; SHUKLA, 2013; WEINHARDT *et al.*, 2015).

Maggiani *et al.* (2013) propõe uma arquitetura para redes de câmeras inteligentes na qual um nó da rede é baseado em microcontrolador, capaz de manipular operações de alto nível e comunicações na rede e um FPGA dedicado ao processamento complexo dos algoritmos de visão computacional. O objetivo foi aproveitar o benefício do paralelismo com o FPGA, permitindo otimizar algoritmos em muito baixo nível (nível de hardware). Os resultados mostraram que a abstração de alto nível provida pela arquitetura não reduziu o desempenho, ao invés disso mostrou uma latência de saída constante e uma solução otimizada, comparado com uma aplicação de software pura. Canis *et al.* (2013) apresenta a ferramenta de síntese de alto nível LegUp que facilita usar técnicas de software para o projeto de hardware. LegUp recebe como entrada um programa em C e compila o mesmo para uma arquitetura híbrida contendo um processador soft MIPS baseado em FPGA e aceleradores de hardware que se comunicam por meio de uma interface de barramento padrão. Os resultados mostraram que a ferramenta produz soluções em hardware com qualidade comparável a ferramentas de síntese de alto nível comerciais. LegUp, juntamente com um conjunto de programas C, é open source e pode ser baixada gratuitamente.

O *framework* SAccO (*Scalable Accelerator platform Osnabrück*) foi proposto para implementação de aplicações que lidam com processamento intensivo de dados, usando aceleradores de hardware reconfigurável (WEINHARDT *et al.*, 2015). O *framework* utiliza PC padrão com placas de FPGA conectadas via *PCI-Express*, em que são implementados os aceleradores. O particionamento das tarefas de uma aplicação a ser executada em paralelo é explorado de forma manual. Para a comunicação e fluxo de dados entre a CPU do PC e o FPGA o *framework* implementa uma API. Além de executar códigos particionados entre a CPU e o FPGA, é possível executar software no PC sem a presença de um FPGA. Contudo, se houver um acelerador em FPGA presente na arquitetura, a API de comunicação transfere dados entre a CPU e o FPGA e, dessa forma, os processos implementados em hardware podem explorar o paralelismo em nível de instrução e *pipeline*. O *framework* possui mecanismos de abstração para “esconder” os FPGAs do desenvolvedor da aplicação. Assim, possibilita selecionar automaticamente o grau ótimo de paralelismo para uma tarefa no FPGA para uma determinada plataforma de hardware. O grau ótimo refere-se a geração de um projeto de hardware que usa a largura de banda disponível

para comunicação entre o PC e o FPGA de forma otimizada.

Adiono *et al.* (2015) citam uma solução para melhorar o desempenho em tempo-real utilizando aceleradores. Aysu, Yuce e Schaumont (2015) apresentam uma técnica de otimização para a geração de assinaturas *lattice-based* em sistemas embarcados. Técnicas de coprojetos hardware/software foram utilizadas. A solução incluiu o particionamento da geração de assinaturas em duas fases, *online* e *offline*, pois grande parte da computação não depende da mensagem estar assinada e dessa forma pode ser manipulada antes da mensagem. Isso possibilita mapear operações de precomputação complexas em software em um processador de baixo custo e utilizar o hardware para acelerar operações *online*. Com esse objetivo duas configurações de projeto foram implementadas. Foi utilizado o FPGA da Altera CGX150 Cyclone-IV. A implementação consistiu em um processador soft core NIOS, um mecanismo de multiplicação *hash* e um multiplicador polinomial. Em média, a arquitetura proposta possibilitou a obtenção de uma melhoria de desempenho de 33 e 105 vezes em comparação com implementações de hardware e software anteriores, respectivamente.

Outro exemplo de aplicação de FPGA como acelerador de hardware é descrito em Neshatpour *et al.* (2015). Uma solução de coprojetos hardware/software é proposta para o algoritmo MapReduce do *framework* Hadoop (HADOOP, 2016) em FPGA. O MapReduce é um modelo de programação usado para lidar com grandes conjuntos de dados em *data centers*. A arquitetura heterogênea (ver Figura 17) consiste de um nó mestre contendo um processador Xeon E5 (INTEL, 2016a) de alto desempenho e um Intel Atom C2758 (INTEL, 2016b) ligado a vários dispositivos Zynq SoCs (XILINX, 2016) como nós escravos. Considerando a sobrecarga de comunicação com o FPGA o ambiente do Hadoop MapReduce aplica técnicas de compressão e descompressão, *random* e ordenação nos dados, segundo (NESHATPOUR *et al.*, 2015). Resultados experimentais revelam um potencial significativo para acelerar os *kernels* de aprendizado de máquina do Hadoop MapReduce usando a metodologia de coprojetos hardware/software.

Yuan *et al.* (2017) propõem uma abordagem de coprojetos hardware/software usando FPGA não volátil e *checkpoint offline/online*. A energia de *backup* é reduzida com a seleção *offline* dos locais de *checkpointing*. Resultados experimentais mostraram que a abordagem proposta reduz em média 39.5% o consumo de energia comparado com outras abordagens similares.

Há diferentes tecnologias para acelerar aplicações e implementar soluções para HPC. No entanto, essa variedade de opções gera um novo desafio: a necessidade de selecionar uma ou mais tecnologias para atingir os objetivos da aplicação. Normalmente, a escolha das tecnologias a serem utilizadas é determinada por fatores, tais como: experiência de uso, familiaridade com alternativas e relação com os fornecedores (DURAN *et al.*, 2014).

Duran *et al.* (2014) apresentam um *framework* que avalia plataformas a serem utilizadas em HPC. Inicialmente é realizada uma análise comparativa sobre as plataformas que consistem de uma CPU multi-core, uma GPU e um FPGA. Elas são avaliadas em contexto hipotético

Figura 17 – Arquitetura do sistema

Fonte: Neshatpour *et al.* (2015).

executando duas aplicações de negócios. O *framework* fornece um processo que avalia quantitativamente as tecnologias atribuindo características e pesos com base na relevância para a aplicação alvo. Uma vez que a avaliação depende de hipóteses que fundamentam as entradas, a avaliação fornece orientações e não uma resposta absoluta sobre as tecnologias mais adequadas. Os elementos para avaliação incluem um conjunto ortogonal de perspectivas (desempenho, custo, tempo, suporte) e seus critérios, uma matriz de avaliação de tecnologia, um conjunto de matrizes de comparação de pares para determinar os critérios e os pesos das perspectivas, um modelo de pontuação ponderada, um mecanismo de revisão de *feedback* e diagramas de radar. A Figura 18 ilustra o processo de avaliação do *framework*.

O *Consortium for Small-scale Modeling* (COSMO) é um modelo atmosférico utilizado para previsão do tempo, desenvolvido e mantido por um consórcio de sete centros meteorológicos dos seguintes países: Alemanha, Suíça, Itália, Grécia, Polônia, Romênia e Rússia. Ele também é usado em universidades e institutos de pesquisa em clima regional. Cumming *et al.* (2013) descrevem o esforço para portar o código fonte do modelo COSMO para diferentes sistemas HPC híbridos da fabricante Cray (CRAY, 2016). O COSMO tem cerca de 250 mil linhas de código em *Fortran 90* com *Message Passing Interface* (MPI). A portabilidade do código teve como objetivo disponibilizar para a comunidade de usuários um sistema que pode ser executado em arquiteturas *many-cores* e nas que utilizam GPUs como aceleradores. As linguagens de programação utilizadas na portabilidade foram C++ e CUDA e as interfaces de programação OpenMP e OpenACC. Para a portabilidade do código original foram adotadas duas abordagens diferentes. A primeira, utilizando uma *Domain-Specific Embedded Language* (DSEL) em C++, compila código OpenMP e CUDA para um arquivo fonte. A segunda adiciona diretivas OpenACC

Figura 18 – Visão geral do processo de avaliação de uma tecnologia

Fonte: Duran *et al.* (2014).

em alguns trechos do código *Fortran 90* original. O resultado obtido em ambas as abordagens foi um *speedup* de execução na faixa de 1.4s – 1.45s nas arquiteturas híbridas (CUMMING *et al.*, 2013).

Em Govil e Chowdhury (2014) a metodologia de coprojeto foi usada para alcançar maior desempenho do sistema e flexibilidade no projeto. O algoritmo foi inicialmente implementado em um microcontrolador, porém havia o problema da lentidão na execução. Com isso optou-se por utilizar um FPGA. O problema passou a ser o limitado número de blocos lógicos configuráveis disponíveis no FPGA. Após essas tentativas uma arquitetura utilizando microcontrolador e FPGA juntos foi proposta. Alvarado e Rosa (2016) propõem uma arquitetura baseada em FPGA para fornecer alto desempenho para processamento de sinais para aplicação em sistemas espaciais.

Allam e Deabes (2016) propõem uma arquitetura reconfigurável baseada em FPGA para o sistema ECT *Electrical Capacitance Tomography*. Nesses sistemas um algoritmo de reconstrução de imagem processa grande quantidade de dados e com a arquitetura proposta os resultados refletiram uma melhoria no desempenho. Jaziri, Charaabi e Jelassi (2017) propõem uma metodologia de coprojeto para um sistema de controle baseado em FPGA. A reconfiguração dinâmica do sistema é fornecida por um servidor web implementado em Beaglebone. Com o objetivo de obter alto desempenho e flexibilidade Fournaris *et al.* (2019) propõem uma abordagem de coprojeto hardware/software para o ECDSA (algoritmo de assinatura digital de curva elíptica). Nessa configuração multiplicação escalar é implementada em hardware e integrada ao ECDSA.

Portar programas/aplicações para serem executadas em FPGA não é uma tarefa trivial. Apesar disso, os atributos positivos do FPGA motivam sua pesquisa e utilização. A ferramenta APPRAISE (*Automated Pipeline for the Parallelisation of Random Scalar Expressions*) e a

máquina Hydra são descritas em [Farrimondand, Collins e Sharma \(2009\)](#). A máquina Hydra é implementada no FPGA e funciona como um interpretador que executa de forma paralela os cálculos implementados no código *Fortran* original transformado pela APPRASE. Para converter o código *Fortran* em um programa a ser executado em FPGA, APPRASE analisa o código fazendo uso do *kit* de ferramentas para *Fortran* WinFPT. A WinFPT é usada para realizar alterações no código *Fortran*. Com o código alterado ele é transformado em instruções para a máquina Hydra. Algoritmos de escalonamento marcam o código a ser executado na Hydra. Por fim, uma última transformação de código é realizada gerando o binário que será executado na Hydra. A abordagem APPRASE viabiliza a migração de um programa inteiro em *Fortran* para um ou mais FPGAs, desde que seja tratado o gargalo de comunicação.

3.3 Trabalhos relacionados ao desempenho do BRAMS

Um estudo para migração de código de máquinas vetórias para *clusters* de micro-processadores IA-32 é apresentado em [Fazenda et al. \(2006\)](#). A técnica aumenta a eficiência na execução de parte do código do BRAMS e combina instruções de vetores com reutilização de memória *cache* e *footprint* da memória. Buscou-se identificar uma estrutura de dados e código parametrizado adequado para ambas as arquiteturas e que executasse bem nos dois casos, com um número mínimo de parâmetros de desempenho. Não era objetivo desse trabalho obter o melhor desempenho possível, mas sim aumentar o nível de desempenho atual em ambos os sistemas e alcançar um grau de portabilidade eficaz.

O BRAMS é executado diariamente em modo de produção em centros de pesquisa no Brasil, utilizando *clusters* de micro-processadores IA-32 e no CPTEC em uma máquina vetorial NEC SX-6 ¹ (Máquina de execução do sistema BRAMS em 2009. Atualmente o BRAMS é executado no Tupã ²). O fragmento de código do BRAMS utilizado neste estudo corresponde a cerca de 12% do tempo total de execução. Ele está organizado como duas sub-rotinas independentes, uma para campos escalares e outra para campos vetoriais, juntas elas possuem cerca de 1100 linhas de código em *Fortran 90*.

Os experimentos foram realizados em um único nó da máquina vetorial NEC SX-6, que contém 8 processadores e 64 Gb de RAM. No *cluster* de micro-processadores IA-32 foi utilizado um único nó contendo um processador *dual core* Intel Xeon de 2.8 GHz, 1 Gb de RAM e 512 Kb de memória *cache* L2. Os dados de entrada correspondem a um *grid* com tamanho horizontal $n \times n$ (tamanho do problema e longitudes e latitudes com o mesmo número de pontos) e 32 níveis verticais.

O primeiro teste consistiu em executar o fragmento de código original sem realizar nenhuma otimização nas duas máquinas. Como resultado, o processador Xeon obteve um

¹ <http://www.cptec.inpe.br/supercomputador/historico.php>

² <http://supercomputacao.inpe.br/recursos2>

desempenho superior para problemas (n) de tamanho pequeno. À medida que o tamanho do problema foi aumentando o NEC SX-6 superou o Xeon. O baixo desempenho do Xeon com o aumento no tamanho do problema é explicado pelo surgimento de gargalos de memória e alta taxa de *cache miss*. Já o baixo desempenho do NEC SX-6 está relacionado com pequeno tamanho do vetor. Como resultado da primeira otimização, houve redução do tempo de execução em relação à execução original. Para o NEC SX-6 houve um ganho de fator 4 na execução, devido aos vetores serem mais longos. No Xeon, para problemas pequenos, também obteve-se aumento da velocidade. Para problemas de tamanho maior, em razão dos vetores mais longos não há ganho sobre o Xeon. Em relação a taxa de *cache miss* dessa solução, se comparada com o teste inicial, não há ganhos expressivos, mesmo com a redução da taxa de *cache miss* nessa otimização.

Na segunda otimização foi aplicada a técnica de *loop tiling* no fragmento de código do BRAMS. Para essa otimização, todos os experimentos foram realizados somente no Xeon. Como resultado, não houve praticamente nenhum ganho em relação ao experimento sem utilizar otimização.

Para a terceira abordagem de otimização, o foco foi um único vetor de dados com o objetivo de reduzir as distâncias de reutilização de memória, conflitos de cache e picos de tempo na execução. Essa abordagem produziu melhor resultado para o Xeon ao trabalhar com problemas maiores.

A quarta abordagem de otimização consistiu em aproximar os dados de vários campos do vetor que são simultaneamente referenciados na memória. Essa foi a abordagem que produziu os melhores ganhos de desempenho para ambas as arquiteturas. Houve um aumento de desempenho de 61% para o NEC SX6 e de 25% para o Xeon, com variação no tamanho do problema. Por fim, o trabalho apresentou um estudo de portabilidade de parte de código do BRAMS para ser executado em duas diferentes arquiteturas de computação. Os ganhos de desempenho obtidos foram alcançados com otimizações em nível de software, alterando o código fonte do fragmento do BRAMS relatado no trabalho.

Uma abordagem para reduzir o tempo de execução de aplicações com um padrão de comunicação constante em *clusters* de processadores *multi-core* é descrito em [Rodrigues et al. \(2009\)](#). Como estudo de caso foi utilizado o sistema BRAMS. Utilizou-se a abordagem *Dual Recursive Bipartitioning*, que consiste em um método estatístico para análise de múltiplas variáveis, dividindo o domínio do problema em sub-domínios.

Nesse trabalho foi realizado um mapeamento estático da comunicação, considerando os diferentes custos de comunicação *inter-cores*, *inter-nó* e *intra-nó* em um computador paralelo. Com a finalidade de estabelecer um parâmetro de comparação foi executada uma aplicação denominada *ping-pong*. Como resultado, a comunicação *inter-nó* foi em média seis vezes mais lenta que a comunicação *intra-nó*. Além disso, a comunicação entre os *cores* que não partilham *cache*, mas estão no mesmo nó, foi em média 50% mais lenta em comparação com os *cores* que

partilham *cache*.

O BRAMS foi escolhido para este estudo, pois de acordo com [Rodrigues et al. \(2009\)](#), ele possui um padrão de comunicação fixa para um problema de tamanho e um número fixo de processadores. Para a realização dos experimentos foram utilizados 36, 49, 64 e 100 processadores dividindo igualmente o domínio do problema para manter o equilíbrio de carga. Como resultado, a abordagem proposta contra o mapeamento padrão usado pelo MPICH, possibilitou uma redução de até 9% sobre o tempo total de execução. Nesse trabalho, a abordagem de mapeamento proposta foi aplicada ao BRAMS. Comparando as execuções utilizando a abordagem e sem utilizá-la para a mesma quantidade de processadores, com a abordagem proposta houve um ganho médio de 6% no tempo de execução, no entanto, à medida que são adicionados núcleos de processamento o ganho tende a diminuir.

O trabalho proposto por [Fazenda et al. \(2009\)](#) investiga como escalar o paralelismo do BRAMS, que executa de forma eficiente em muitas dezenas de processadores, para máquinas com muitas centenas de processadores. Esse estudo relata alterações no BRAMS para fazer uso de uma máquina com muitas centenas de processadores. Os experimentos mostram que o aumento no número de processadores possui uma estreita relação com as fases de computação realizadas no BRAMS, uma vez que ocorrem mudanças de gargalos na aplicação à medida que aumenta-se o número de processadores para execução do BRAMS. Para os experimentos da abordagem proposta foram utilizados até 700 processadores em *grid*, com resolução horizontal de 20 e 10 km sobre a totalidade da América Latina. Ainda de acordo com [Fazenda et al. \(2009\)](#), o comportamento do BRAMS, quando executado em centenas de processadores é desconhecido. Tipicamente, o BRAMS é executado em dezenas de processadores, em previsões com uma resolução de 80 ou 40 km, ou em domínios geográficos muito menores (poucos km).

Para uma previsão de período fixo em uma grade de dimensão horizontal, o BRAMS possui uma complexidade computacional cúbica ([FAZENDA et al., 2009](#)). Ao refinar a resolução do modelo de 20 para 10 km, por exemplo, o tempo de execução é aumentado em aproximadamente 8 vezes. Esse aumento no tempo de execução justifica o acréscimo no número de processadores. Para medições iniciais de desempenho utilizou-se a versão 4.1 do BRAMS em previsões de 24 horas para as resoluções de 10 e 20 km.

O paralelismo implementado no BRAMS é do tipo mestre-escravo e o sistema computacional utilizado foi o *cluster* UNA do INPE/CPTEC. Ele é composto por 275 nós, cada um com 8GB de memória principal e dois processadores AMD Opteron *dual core* (Opteron 2218), totalizando 1100 *cores*. O BRAMS possuiu 5 fases da computação, são elas: inicialização, condições de contorno, *timestep*, CFL (estabilidade numérica de Courant, Fridrichs e Lewy) e a saída. De acordo com [Fazenda et al. \(2009\)](#), o paralelismo no BRAMS é concentrado no trecho mais demandante, o *timestep*. Isso ocorre devido a quantidade de operações de ponto flutuante por ele executada, as demais fases executam poucas operações ponto flutuante. No entanto, esse trecho não é o maior responsável por limitar a escalabilidade do BRAMS.

Como conclusão dos experimentos, ficou evidente os limites de escalabilidade, pois para resoluções 10 e 20 km há um número ótimo de processadores. No entanto, na execução para uma resolução de 10 km ocorre paginação pelo processo mestre, fato que não ocorre para a resolução de 20 km. Esse comportamento ocorre, pois o processo mestre armazena os campos meteorológicos no domínio completo, enquanto que os escravos armazenam apenas sua partição de domínio. Segundo [Fazenda et al. \(2009\)](#), a computação estruturada dessa forma é adequada para execuções pouco demandantes em máquinas com poucos processadores, porém, ela não é indicada para execuções em máquinas como o UNA.

Com a finalidade de identificar as fases de computação do BRAMS limitantes da escalabilidade, o código fonte foi instrumentado para medir o tempo de execução de cada fase. Na execução desse experimento, foi utilizada apenas a resolução de 20 km. Como resultado desse experimento, as fases inicialização, condição de contorno e estabilidade numérica (CFL), que apresentavam um baixo tempo de execução até 200 processadores, obtiveram um aumento significativo a partir de 300 processadores. Na fase *timestep* ocorreu o inverso. De acordo com [Fazenda et al. \(2009\)](#), os algoritmos utilizados nessas fases possuem trechos sequenciais e há uma troca de grandes volumes de dados entre o mestre e os escravos visando I/O, por isso ocorreu esse comportamento. Como solução para os problemas encontrados, foi feita uma reestruturação e implementação paralela do código de cada uma das fases limitantes da escalabilidade, visando reduzir a competição por I/O entre os processos escravos e utilização de memória em excesso.

A transformação dos gargalos sequenciais em algoritmos paralelos reduziu o trabalho do processo mestre, uma vez que ele propaga as informações para inicialização da computação e em seguida efetua a mesma computação que os escravos, entretanto, leituras em disco são realizadas apenas pelo mestre. Com as alterações, o modelo BRAMS possui aproximadamente 160.000 linhas em *Fortran 90* e algumas centenas em *C*. Comparando-se com a versão original as alterações atingem 30% do total.

Com as alterações o BRAMS foi escalado para ser executado em centenas de processadores, pois, foi obtido um ganho de desempenho com 200 e 300 processadores, seguido de uma queda de eficiência utilizando de 400 a 600, tomando como base o tempo de processamento com 100 processadores. Além disso, o trabalho aponta o desbalanceamento de carga como um fator limitante, que é um problema pouco explorado em modelos meteorológicos.

O trabalho descrito em [Rodrigues et al. \(2010\)](#) apresenta um estudo conduzido no BRAMS para balanceamento de carga entre processadores. O estudo apresenta uma análise comparativa entre diferentes algoritmos de balanceamento de carga para lidar com o desequilíbrio dinâmico causado por variações localizadas no estado da atmosfera. As análises do estudo consideraram o impacto sobre a computação e comunicação entre os processadores e os efeitos do balanceador de carga, à medida que o mesmo era chamado em tempo de execução.

Segundo [Rodrigues et al. \(2010\)](#), tempestades, processos químicos e biológicos (queima de biomassa, por exemplo) são fontes que causam desequilíbrio de carga durante a execução de

modelos de previsão meteorológica. Além disso, existe um número limitado de pesquisas em estratégias de balanceamento de carga em modelos meteorológicos. Porém, essas pesquisas não são aplicadas em códigos em produção, uma vez que não é trivial implementar balanceamento de carga em códigos legados. Nesse trabalho, foram usados processadores virtuais e físicos. O balanceador de carga foi inserido em um ambiente virtual MPI para separar o algoritmo de balanceamento da aplicação. Nesse esquema, o domínio tratado pela aplicação foi dividido em sub-domínios, nos quais a quantidade de sub-domínios foi superior ao número de processadores físicos. Para cada processador físico foi atribuído um conjunto de processadores virtuais e para cada um deles foi atribuído um subdomínio.

Balanceamento de carga em modelos de tempo e clima tem sido reconhecidamente um problema a ser pesquisado (RODRIGUES *et al.*, 2010). Além disso, não há um algoritmo de balanceamento de carga dinâmico para esses modelos em que o código seja independente do código do modelo da aplicação. No BRAMS, como em outros modelos de previsão meteorológica, a atmosfera é representada com uma grade tridimensional de pontos, na qual cada ponto é calculado por uma *thread*. Dessa forma, no mapeamento das *threads* no BRAMS, duas *threads* de domínios vizinhos estão situadas no mesmo processador físico, com o propósito de garantir para elas uma comunicação local, minimizando a sobrecarga de comunicação. A solução proposta para comparar os algoritmos de balanceamento aplicados nesse trabalho consistiu inicialmente em observar as *threads* que cobrem o domínio com uma curva de Hilbert e modificar a curva de forma recursiva de acordo com as cargas observadas nas *threads*. Esse processo é repetido até igualar o número de segmentos do domínio com o número de processadores físicos, de modo que, os segmentos possuam a mesma carga de trabalho e também regiões vizinhas para preservar a relação espacial entre as *threads*.

Nos experimentos foram utilizados 64 processadores físicos e 1024 processadores virtuais. Como resultado do primeiro experimento, os balanceadores de carga produziram um melhor desempenho em comparação com o caso em que não foram utilizados processadores virtualizados. No entanto, o excesso de decomposição ocasionou perda de parte do desempenho, comparando-se ao caso em que não foi utilizado um balanceador de carga. Além disso, o custo de execução do algoritmo de equilíbrio e os custos de migração das *threads* foram excessivos. O balanceador não foi capaz de reequilibrar a carga completamente e com isso a comunicação inter-processador aumentou após o reequilíbrio.

Com base nos resultados do primeiro experimento foi proposta uma estratégia de equilíbrio adaptativo, pois segundo Rodrigues *et al.* (2010), o custo de execução do algoritmo de balanceamento de carga é geralmente baixo, e o custo de migração de *threads* é elevado. Com isso, o balanceador de carga é invocado com mais frequência e as migrações só ocorrem se um determinado limite de desequilíbrio é observado.

Para realização dos experimentos com equilíbrio adaptativo foram utilizados os limites de 50%, 20%, 10% e 5% de desequilíbrio para desencadear as migrações das *threads*. De modo

resumido, existe um ponto ótimo para o esquema de equilíbrio adaptativo, que é atingido quando a frequência de invocação do balanceador de carga é alta o bastante para detectar as variações de carga. Esse fato ocorreu quando o desequilíbrio foi superior a 10% e resultou em um ganho de desempenho de 37,3% em relação a uma execução do BRAMS sem virtualização.

O trabalho proposto por [BERNARDI et al. \(2013\)](#) consistiu em paralelizar uma sub-rotina do BRAMS utilizando *OpenCL*. A sub-rotina escolhida possui cerca de 220 linhas de código em *Fortran 90*, e sua estrutura dispõe de loops aninhados. Esse tipo de estrutura normalmente é bem explorada para paralelismo com grande número de threads. Segundo [BERNARDI et al. \(2013\)](#), o tempo médio de execução dessa sub-rotina é de 36,79 ms, tempo que é considerado relativamente rápido para o estudo em questão. No entanto, a motivação em melhorar o desempenho de execução dessa sub-rotina é a grande quantidade de vezes que ela é chamada. O código da sub-rotina foi adaptado para possibilitar a execução do código *Fortran* em conjunto com código *OpenCL*. Após a análise e passagem dos parâmetros da sub-rotina, a representação e a forma de acesso dos vetores em *Fortran* para *OpenCL*, para ser executado na GPU o kernel da sub-rotina foi codificado e instrumentado para a realização dos experimentos. O ambiente utilizado foi um processador Intel Xeon E5620 2,4Ghz, 12 GB DDR3 de RAM e uma GPU Nvidia Tesla M2050 3GB GDDR5. A realização dos experimentos consistiu em uma média de 23 execuções, para um período de previsão do tempo de 24 horas. Como resultado dos experimentos, observou-se que a sub-rotina adaptada para *OpenCL* consumiu a maior parte do tempo em inicializações, criação e liberação de *buffers* de memória na GPU. Isso resultou no desempenho de execução paralela da sub-rotina inferior ao da implementação serial. Como conclusão, a implementação *OpenCL* desenvolvida foi ineficiente para a paralelização da sub-rotina escolhida ([BERNARDI et al., 2013](#)).

Em [Evangelista, Fazenda e Melo \(2014\)](#) é apresentado um estudo utilizando GPU para otimizar o desempenho de um trecho de código do BRAMS, implementando-o como um sistema dataflow. Para aplicar dataflow foi adotada uma abordagem capaz de identificar sequências independentes de dados contíguos que podem ser processados em paralelo. Além da identificação das sequências foi desenvolvido um método para classificação dos dados que permite identificar o tipo de processamento a ser executado e suas possíveis dependências com os elementos adjacentes. Nesse trabalho foi desenvolvido um algoritmo de parte do código do BRAMS para GPU como estudo de caso, que resolve inicialmente todas as tarefas independentes e posteriormente as que apresentam dependência. Dessa forma, as tarefas são mapeadas em uma lista, de modo que as tarefas independentes são agrupadas em sequência, logo após as tarefas dependentes. Contudo, segundo [Evangelista, Fazenda e Melo \(2014\)](#), o conjunto de tarefas independentes gerado é heterogêneo quanto ao esforço computacional e fluxo de execução e isso impõe um padrão irregular de acesso à memória por tarefa. Em GPU, esse acesso à memória é um desafio para obter eficiência computacional.

Foram realizadas alterações na estrutura de dados, no algoritmo, na configuração lógica

da estrutura de dados na GPU e nas diretivas OpenACC usadas na programação. Tais alterações concentram-se em: 1) código em OpenACC não otimizado; 2) ajustes de uso de blocos e vetores na GPU; 3) otimização de transferência de dados; 4) alteração algorítmica para adaptação a GPU e; 5) redução do uso de *branches*. Como resultado, a redução no uso de *branches* causou uma melhoria no acesso de leitura à memória global e a modificação algorítmica proporcionou uma melhor taxa de ocupação na GPU, o que refletiu em uma melhoria na eficiência de acesso de escrita à memória global.

Logo, conclui-se que apenas introduzir diretivas elementares do OpenAcc no código não produz um código eficiente para GPU. Foram necessárias diversas alterações e especificações de diretivas auxiliares em OpenAcc e alterações no algoritmo. Houve um ganho de desempenho do algoritmo em GPU se comparado a sua execução em CPU, no entanto, de acordo com [Evangelista, Fazenda e Melo \(2014\)](#) o desempenho obtido pode ser considerado modesto, comparando-o com otimizações em outros métodos que o trecho de código do BRAMS estudado implementa. Como exemplo, métodos tradicionais para dinâmica de fluídos computacional. Porém, para o algoritmo desenvolvido a solução por dataflow atingiu um valor que pode ser considerado alto para uso em GPU.

O módulo da química do BRAMS possui um termo na equação da continuidade de massa, que é referente a atualização da composição química da atmosfera. Este termo, produz um sistema não linear de equações diferenciais, que ao ser discretizado, implica na resolução de um sistema algébrico de equações lineares, esparsas e rígido. A resolução desse sistema de equações é computacionalmente custosa para todos os pontos da grid a cada passo de tempo com o aumento da resolução horizontal ([FERNANDES, 2014](#)).

[Fernandes \(2014\)](#) propõe a utilização de um solver paralelizável para a resolução das equações lineares diferenciais do módulo da química do sistema BRAMS. O solver utilizado é implementado de forma sequencial e o seu uso de forma operacional é custoso. Dessa forma, foi proposto e implementado um solver dessas equações baseado em estimação linear ótima, que possui um código mais simples e mais apropriado à paralelização.

O solver original usado no BRAMS é implementado na linguagem C, sendo extremamente rápido e estável na solução do sistema de equações. Entretanto, as otimizações incluídas na programação de suas rotinas, tais como alocação dinâmica da memória e o uso de ponteiros para listas encadeadas, dificultam a paralelização. Essas rotinas correspondem a quase 13.000 linhas de código e foram compiladas e incorporadas ao BRAMS, entretanto, todo o restante do módulo químico é codificado em *Fortran 90*, somando mais de 27.000 linhas de código. A paralelização “interna” do solver original fica inviável pela complexidade do código e baixa granularidade. Além disso, o fato do módulo da química estar codificado em duas linguagens de programação impede o uso do OpenMP, pois, em geral, compiladores OpenMP não permitem chamadas a funções sem fazer *inlining* das mesmas, o que naturalmente impede o uso de funções em linguagens diferentes ([FERNANDES, 2014](#)).

Em testes realizados na execução do BRAMS utilizando o solver original no módulo da química do BRAMS, observou-se um aumento de 100 vezes no tempo de processamento em relação ao não uso do mesmo. O solver proposto e implementado (2 versões) nesse trabalho foi codificado em *Fortran 90* e possui 60 linhas de código. De acordo com [Fernandes \(2014\)](#), os testes realizados com esse solver obtiveram resultados numericamente equivalentes ao solver original sequencial. Uma comparação feita entre as implementações sequencias dos solvers mostrou que o novo solver demanda cerca de 35% de acréscimo de tempo, comparando com o solver original. No entanto, vale destacar que o solver proposto não foi integrado ao BRAMS. Os testes foram realizados de modo *stand alone*.

Nos testes com as implementações paralelas com diretivas OpenMP do novo solver, observou-se que ocorre um ganho de desempenho utilizando 4 *threads*. Acima dessa quantidade houve perda de desempenho para a versão 1 da implementação. Para a versão 2 do solver o melhor desempenho foi obtido utilizando 32 *threads*. Para a versão 2 do solver com 32 *threads* o tempo de execução do módulo foi reduzido em 16 vezes em relação ao tempo sequencial. Verificou-se também a possibilidade de portar o novo solver para ser executado em GPU vislumbrando um futuro supercomputador no CPTEC/INPE que dispusesse de tal dispositivo, no entanto, os resultados para essa plataforma não foram satisfatórios.

Em [Souza, Pereira e Marques \(2017\)](#) é descrito um coprojeto hardware/software usando FPGA para o solver do módulo da química do BRAMS. O estudo reportado consistiu em usar *OpenCL* para implementar o método Jacobi em hardware para resolução de equações lineares do tipo $Ax = b$. Nesse trabalho o solver original do BRAMS foi substituído pela implementação em hardware e utilizado o modelo de programação Task e NDRange do *OpenCL*. O BRAMS foi integrado com o solver implementado em hardware e o sistema foi executado em uma máquina heterogênea com CPU e FPGA conectado ao sistema via barramento PCIexpress. Os resultados mostraram que o tempo de execução foi em média 18 vezes mais lento em relação a execução do sistema com o solver original em software. Para esse tipo de problema a programação em Task apresentou melhores resultados do que a programação em NDRange.

Em [Carreño et al. \(2015\)](#) é descrito um estudo da execução do BRAMS com uma aplicação de computação científica para previsão do tempo, no ambiente de Cloud Computing Microsoft Azure. Os experimentos consideraram o escalonamento do número de nós disponíveis para executar o BRAMS em uma previsão do tempo de 72 horas, com uma resolução espacial de 20 e 5 km, cobrindo uma área de 1500x1500 km. Para cada resolução, a previsão foi realizada cinco vezes em máquinas virtuais com 8 *cores* de CPU.

O estudo teve como objetivo avaliar a escalabilidade no número de máquinas virtuais utilizadas para executar o BRAMS, considerando a quantidade total de processadores envolvidos. O processo de adição de máquinas virtuais foi repetido até atingir um limite em que a adição de máquinas não influenciava no aumento de desempenho e sim o custo do experimento em adquirir máquinas virtuais. Como resultados, observou-se que máquinas virtuais semelhantes (diferença

apenas no número de cores de CPU) produziram resultados diferentes quanto a escalabilidade, pois máquinas com menos cores de CPU produziram resultados melhores que máquinas com mais cores de CPU. Além disso, máquinas com mais cores custam o dobro das máquinas com menos cores. Com os resultados, ficou evidente que no caso do BRAMS, uma máquina virtual duas vezes mais cara não é melhor na mesma ordem de grandeza em termos de desempenho.

Por fim, no estudo concluiu-se que é possível escalar e utilizar um ambiente de *cloud computing* para executar o BRAMS. Contudo, a escala para esse ambiente foi de até 168 núcleos de CPU. Após o uso de 88 núcleos a redução do tempo de execução é inferior a 2% a cada aumento no número de núcleos. Tratando-se de um ambiente virtual, segundo [Carreño et al. \(2015\)](#), a escalabilidade alcançada não é totalmente confiável, e assim, são necessários testes em um *cluster* físico para confirmar os resultados obtidos.

O Quadro 1 ilustra as principais características dos trabalhos relacionados. A coluna **Otimização** da tabela refere-se se a otimização necessária para obter um ganho de desempenho ou escalabilidade na execução do BRAMS. Ela pode ser realizada a nível de software ou hardware (neste contexto significa aumento do número de cores de processamento utilizado e/ou mudança de tecnologia do processador). A coluna **Ambiente** refere-se a execução para ganho de desempenho e/ou escalabilidade na execução do BRAMS em máquinas físicas ou virtualizadas. A coluna **Plataforma de execução** refere-se a unidade de processamento utilizada na execução do BRAMS ou a trechos de código do mesmo. A coluna **I/O** refere-se a considerar ou apontar o impacto das operações de I/O como um gargalo para o ganho de desempenho ou escalabilidade do BRAMS na execução realizada no trabalho. A coluna **Balanceamento** representa se o estudo considera o desbalanceamento de carga como um gargalo para o ganho de desempenho ou escalabilidade do BRAMS na execução realizada no trabalho. E, por fim, a coluna BRAMS está realacionado se o trabalho utilizou o código BRAMS de forma completa ou parcial (módulo ou rotinas do código).

A Figura 19 apresenta um gráfico de bolhas para os trabalhos relacionados ao estudo de desempenho do BRAMS, de acordo com as características apresentadas no Quadro 1.

Quadro 1 – Trabalhos relacionados ao estudo de desempenho do BRAMS

Ano	Autor	Otimização	Ambiente	Plataforma	I/O	Balanceamento	BRAMS
2006	Fazenda et al.	SW	Real	CPU	Não	Não	Completo
2009	Rodrigues et al.	SW	Real	CPU	Sim	Sim	Completo
2009	Fazenda et al.	HW/SW	Real	CPU	Sim	Sim	Completo
2010	Rodrigues et al.	SW	Virtual/Real	CPU	Não	Sim	Completo
2012	Fazenda et al.	HW	Real	CPU	Sim	Não	Completo
2013	Bernadi et al.	SW	Real	CPU/GPU	Não	Não	Rotina
2014	Evangelista et al.	SW	Real	GPU	Não	Não	Rotina
2014	Fernandes	SW	Real	CPU/GPU	Não	Não	Rotina
2014	Fazenda et al.	SW	Virtual	GPU	Não	Sim	Rotina
2015	Carreño et al.	HW	Virtual	CPU	Não	Não	Completo
2015	Silva Junior, M. B.	SW	Real	CPU/GPU	Não	Não	Módulo
2017	Souza et al.	HW	Real	CPU/FPGA	Não	Não	Rotina
2018	Pereira, E.S.	HW	Real	CPU/FPGA	Não	Não	Módulo

Figura 19 – Publicações relacionadas ao estudo de desempenho do BRAMS

Fonte: Elaborada pelo autor.

METODOLOGIA

4.1 Considerações Iniciais

Neste capítulo é apresentada a metodologia utilizada para o desenvolvimento deste trabalho de doutorado, considerando os conceitos apresentados nos capítulos anteriores.

Este capítulo está organizado da maneira a seguir. Na Seção 4.2 é apresentada a metodologia utilizada para o desenvolvimento deste doutorado. Na Seção 4.3 é descrito o profile do módulo da dinâmica do BRAMS.

4.2 Descrição das etapas

A metodologia para desenvolvimento e avaliação desta proposta de trabalho incluiu inicialmente um estudo sobre as tecnologias que seriam utilizadas no desenvolvimento do projeto. Nesta etapa foram estudadas tecnologias envolvendo técnicas de *profile* e ferramentas para geração de hardware que aceitam entrada de código *C* ou *Fortran 90*. Outro ponto estudado foi a estrutura física e lógica necessária para o desenvolvimento do *framework* proposto e formas de avaliar o mesmo.

O passo seguinte foi a seleção de parâmetros e métricas de avaliação. Nesta etapa foram definidos, observando o estado da arte, os parâmetros e métricas a serem selecionadas para possíveis comparações com trabalhos semelhantes, com o objetivo de avaliar o desempenho da arquitetura gerada pelo *framework* proposto. Nesta etapa contamos com o apoio dos pesquisadores e desenvolvedores do BRAMS no CPTEC/INPE que nos forneceram detalhes de implementação do sistema bem como informações de como a previsão do tempo é realizada.

Com essas tarefas em andamento ocorreu a preparação do ambiente de trabalho e geração de *profiles* do módulo da dinâmica do BRAMS. Nesta etapa foram instaladas as ferramentas para o desenvolvimento do trabalho em três diferentes plataformas. A primeira plataforma foi

o *kit* de desenvolvimento DE2i-150, que possui um processador Atom *dual core* e um FPGA Cyclone IV da Altera. A segunda plataforma possui a seguinte configuração: uma máquina com um processador Xeon E5-1607 v3 e conectado via barramento PCI express um FPGA Stratix V (BittWare Board S5PH-Q). Por fim, a terceira plataforma foi o cluster HARP disponibilizado pela Intel, hospedado na Universidade de Paderborn, na Alemanha.

Os diferentes ambientes serviram para confirmar os gargalos do módulo da dinâmica do BRAMS, uma vez que no CPTEC/INPE ele é executado em um *cluster* contendo centenas de *cores*. Em outros centros de pesquisa do Brasil ele é executado em máquinas contendo dezenas de *cores*. Esta proposta teve como objetivo, além de medir o desempenho das arquiteturas geradas pelo *framework*, disponibilizar uma metodologia de geração de hardware para outros módulos do BRAMS usando FPGA.

Uma vez preparado o ambiente de trabalho, foram gerados os *profiles* de execução da dinâmica do BRAMS nos ambientes apresentados nesta seção com a finalidade de encontrar gargalos com um alto grau de similaridade entre as plataformas. Esperava-se nesta fase que os tempos de execução diferissem para as plataformas e que os gargalos permanecessem os mesmos. Apesar da execução nas três plataformas optou-se, devido a tendência tecnológica e melhores recursos, que as fases de codificação e prototipação fossem direcionadas ao HARP.

Na fase de prototipação e testes foi criado um protótipo funcional do *framework*. O protótipo foi testado, buscando verificar se ele atendia a proposta deste trabalho, no sentido de gerar arquiteturas de hardware reconfigurável para melhorar o desempenho de execução do módulo da dinâmica do BRAMS.

Os experimentos consideraram diferentes módulos de hardware executando em conjunto com a dinâmica do BRAMS, os dados utilizados e a resolução usada na previsão do tempo.

Os dados foram analisados considerando 30 execuções para cada kernel. A ênfase dada nessa etapa consistiu em observar o tempo de execução do módulo da dinâmica nas execuções utilizando apenas a CPU e com o coprojeto de hardware e software proposto nesta tese para o módulo da dinâmica do BRAMS.

A Figura 20 apresenta um fluxograma simplificado das etapas da metodologia adotada para o desenvolvimento deste trabalho de doutorado.

Figura 20 – Fluxograma simplificado da metodologia

Fonte: Elaborada pelo autor.

4.2.1 Kit de desenvolvimento DE2i150

O *kit* DE2i-150 é uma plataforma de desenvolvimento que combina de forma embarcada um processador Intel Atom dual core modelo N2600 com a flexibilidade de um FPGA Altera Cyclone IV GX. O processador e o FPGA são interligados via barramento *PCI Express* (PCIe), que fornece uma comunicação de alta velocidade entre eles (TERASIC, 2015).

Ao combinar um processador de propósito geral com um FPGA, a plataforma oferece um poderoso ambiente para desenvolvimento de projetos que combinem soluções de hardware e software. O FPGA Cyclone IV, presente na placa, possui 149.760 elementos lógicos e pode ser utilizado como um coprocessador, visando acelerar o tempo de resposta de uma aplicação e, ao mesmo tempo, manter um baixo consumo de energia.

Além de estar disponível no Laboratório de Computação Reconfigurável (LCR) do ICMC/USP, A utilização deste *kit* de desenvolvimento foi motivada pelas informações divulgadas pela Intel por meio do programa HARP sobre a integração de um processador Xeon com um FPGA Arria 10 em um mesmo *chip*. Apesar de no *kit* de desenvolvimento DE2i-150 o processador e o FPGA estarem separados, os testes conduzidos na DE2i-150 serviram de base para as implementações na plataforma anunciada pela Intel.

A Figura 21 ilustra o diagrama de blocos do *kit* de desenvolvimento DE2i-150, no qual é possível visualizar as interfaces de comunicação entre o Atom e o FPGA, assim como todo o ferramental disponível na placa.

Figura 21 – Diagrama de bloco do *kit* de desenvolvimento DE2i150

Fonte: Terasic (2015).

O *kit* DE2i-150 foi utilizado na fase inicial do projeto com o objetivo de entender como funcionava a interface de comunicação entre o processador Intel Atom e o FPGA Cyclone. Pretendia-se executar o sistema BRAMS em conjunto com os blocos de hardware a serem desenvolvidos em mais duas plataformas de computação com um certo grau de similaridade com a plataforma DE2i-150.

Outro ponto que também foi trabalhado inicialmente nessa plataforma foi a execução de código *Fortran* em FPGA e/ou a integração do sistema BRAMS, escrito em *Fortran*, com o hardware programado dos *kernels* a serem desenvolvidos no FPGA. As pesquisas realizadas até o momento não revelaram trabalhos concluídos ou em andamento que integram as duas tecnologias.

Desse modo, foi realizado um teste de comunicação CPU-FPGA executando o projeto disponível em (SCHAUMONT, 2013). O projeto foi desenvolvido na forma de um tutorial para um curso de graduação realizado na Faculdade Virginia Tech, Virginia, Estados Unidos, utilizando a placa DE2i-150.

O ambiente da placa no qual o projeto foi desenvolvido é o original fornecido pela fabricante Terasic executando o sistema operacional Yocto¹. Para a condução do teste foi

¹ <https://www.yoctoproject.org/>

necessário modificar o Sistema Operacional (SO) Yocto, uma vez que esse SO possui algumas limitações e não possuía as condições necessárias para execução do código BRAMS, como pacotes, compiladores, etc.

Foi instalado o sistema operacional Ubuntu Server para adequação às necessidades do projeto em razão do suporte existente e da ampla comunidade de usuários. Durante esse processo surgiram algumas dificuldades e problemas com o *driver* do IP *PCI Express* disponibilizado no projeto, pois ele não funcionava corretamente no novo ambiente configurado para execução do projeto.

Além disso, o código fonte do driver fornecido pela fabricante Terasic não está disponível no CD de softwares e exemplos que acompanham o *kit*, nem na página oficial na Internet. Este problema foi solucionado com o suporte da Terasic (Assinatura de cláusulas de confidencialidade) solicitando os fontes do *driver* para que o mesmo pudesse ser alterado e compilado para o ambiente de desenvolvimento com o SO Ubuntu Server instalado. Com as alterações foi possível compilar e executar o projeto.

Para a condução do teste de comunicação foi utilizado um programa em C que lê a posição dos 16 *switches* presentes na placa DE2i-150 e exibe a configuração selecionada dos *switches* como um valor hexadecimal nos *displays* de 7 segmentos. Os periféricos (*switches* e *displays*) são controlados por meio do FPGA, e o acesso aos seus valores (leitura e escrita no caso dos *displays*) é realizado a partir do processador Atom. A execução desse teste foi concluída com sucesso, pois a execução do programa no Atom produziu as saídas esperadas e permitiu verificar a comunicação via *PCI Express* entre a CPU e o FPGA de forma prática.

4.2.2 Usando o FPGA como coprocessador

Neste teste foi demonstrada a utilização do FPGA como um coprocessador executando o algoritmo de criptografia *Advanced Encryption Standard* (AES). O projeto para realização desse teste foi desenvolvido por (YALI; SCHAUMONT, 2013) de forma simples e genérica, com o objetivo de ser compreensível e reutilizável para implementações mais otimizadas.

Uma representação em alto nível do projeto é ilustrada na Figura 22. De acordo com a figura é possível observar que no lado do software há uma aplicação em C que se comunica com o lado do hardware através do *PCI Express*. Também é utilizada a função *nmap*, que faz parte do *kernel* do SO Linux, para mapear uma região de memória física do FPGA para um intervalo de endereços no espaço do usuário. Este é utilizado pela aplicação executada na CPU. Já do lado do hardware, o compilador no ambiente do Quartus II, fornecido pela Altera, é responsável por traduzir uma interface PCI em uma interface Avalon, tornando possível a integração do IP PCI com outros componentes através do barramento Avalon.

O hardware também dispõe de uma memória *on-chip* que pode ser usada de forma compartilhada para trocar dados entre o software e o hardware. Há também duas portas I/O

paralelas que podem ser usados para a comunicação direta entre o programa *C* e o bloco de hardware que calcula o AES.

A realização desse experimento foi importante para este projeto de doutorado, uma vez que possibilitou de forma prática iniciar os estudos e práticas necessárias à construção da biblioteca de hardware desenvolvida durante a execução deste projeto.

Além desses experimentos práticos que foram realizados com programas escritos em *C*, também foi realizado um teste, no qual o código *C* invoca rotinas em *Fortran* executando no processador Atom. O mesmo foi feito para um código *Fortran* executando no Atom, invocando rotinas em um código *C*, que comunica e utiliza o módulo de cálculo do AES gravado no FPGA.

Esse teste teve o intuito de avaliar como seria a execução da aplicação final que integra código *Fortran* sendo executado na CPU e os blocos de hardware presentes no FPGA desenvolvidos em *OpenCL*. *OpenCL* e *Fortran* não “conversam” de forma direta, havendo a necessidade de incluir uma camada de comunicação entre a aplicação em *Fortran* e o hardware desenvolvido em *OpenCL*.

Figura 22 – Diagrama de blocos da arquitetura do coprojeto

Fonte: Yali e Schaumont (2013).

4.2.3 Placa BittWare S5PH-Q

Outra plataforma de FPGA que foi utilizada no desenvolvimento deste projeto foi a placa da BittWare modelo S5PH-Q, a qual possui um FPGA Altera Stratix V com 952.000 elementos lógicos. Essa plataforma foi conectada via barramento *PCI Express* a placa-mãe de um computador disponível no LCR do ICMC/USP com a seguinte configuração:

- Processador: Xeon E5-1607 v3;
- Memória RAM: 16GB;
- Sistema Operacional: Ubuntu 14.04;
- FPGA: Stratix V (BittWare Board S5PH-Q).

De acordo com a [BittWare \(2015\)](#) a BittWare S5PH-Q é uma placa do tipo PCIe x8, baseada no Altera Stratix V GX ou no GS FPGA, que possuiu uma grande largura de banda e baixo consumo de energia. A placa foi projetada com o objetivo de atender aplicativos “de ponta”, uma vez que dispõe de um FPGA Stratix V, mais de 16 GBytes de memória, que incluem DDR3 e QDRII/II +, e duas interfaces QSFP para processamento em rede de alto desempenho. Além disso, o Stratix V disponível na BittWare S5PH-Q suporta a reconfiguração dinâmica, enquanto outras partes do projeto de hardware gravado no FPGA estão sendo executadas.

Além de estar disponível no LCR essa placa de desenvolvimento foi utilizada neste projeto, pois ela possui mais elementos lógicos do que a DE2i-150. A Bittware S5PH-Q é um FPGA moderno com características e elementos de hardware que podem ser usados para acelerar aplicações de computação de alto desempenho, como é o caso da dinâmica do BRAMS, por exemplo. A Figura 23 ilustra o diagrama de blocos da placa S5PH-Q.

Figura 23 – Diagrama de blocos da placa BittWare S5PH-Q

Fonte: [BittWare \(2015\)](#).

Após a experiência adquirida com os testes realizados na plataforma DE2i-150, foi desenvolvido um kernel em *OpenCL* para ser executado na BittWare S5PH-Q. O kernel desenvolvido consistiu em enviar 5 matrizes com as mesmas dimensões processadas pelas rotinas do BRAMS para o FPGA Stratix V, modificar o valor dessas matrizes e devolver para o programa host executando no processador Xeon. O teste foi bem-sucedido, no entanto o tempo de cópia das matrizes foi alto, bem como o tempo de processamento. Os resultados obtidos nesse teste, em grande parte, foram influenciados pela pouca experiência com a linguagem Intel *OpenCL*.

4.3 Profile do módulo da dinâmica do BRAMS

Após instalado e configurado o ambiente de trabalho foram iniciados os preparativos para selecionar os trechos de código do módulo da dinâmica do BRAMS que seriam implementados em *OpenCL* e posteriormente sintetizados para o FPGA. Gerar o profile do código a ser feito o coprojeto hardware/software é uma etapa importante, pois através dele é possível identificar os trechos de código que demandam a computação (gargalos).

Nesse sentido, a execução do módulo da dinâmica foi realizada nos três ambientes de computação inicialmente estabelecidos com a finalidade de encontrar e/ou apontar que independente da plataforma de computação heterogênea contendo CPU e FPGA os gargalos são os mesmos.

A Figura 24 ilustra o resultado do profile executado no HARP. Ao observar a Figura 24 é possível visualizar os pontos na execução da dinâmica que dominam a computação. Na sequência, foi gerado outro profile que permitiu observar dentro das funções apresentadas na Figura 24 quais linhas dominavam a computação quando a função era executada. Em alguns casos, como por exemplo, as funções **gradyv_**, **gradyt_**, **gradxu_** e **gradxt_**, foram implementadas de forma completa em hardware.

ESPECIFICAÇÃO DO *FRAMEWORK*

5.1 Considerações Iniciais

Este capítulo apresenta um coprojeto hardware/software para o problema abordado nesta tese: portar código do BRAMS para um ambiente de computação heterogênea composto por CPU e FPGA. A solução para o problema consiste em desenvolver um *framework* para coprojeto de hardware/software de parte do modelo climático BRAMS para uma máquina heterogênea que utiliza CPU e FPGA.

Assim, o capítulo está organizado da seguinte forma. A Seção 5.2 descreve um estudo do código fonte da dinâmica do BRAMS conduzido para implementação da biblioteca de hardware que compõe o *framework* proposto nesta tese de doutorado. A Seção 5.3 detalha o *framework* para coprojeto de hardware/software do módulo da dinâmica do BRAMS.

5.2 Código-fonte do módulo da Dinâmica

Inicialmente, pretendia-se trabalhar com o código fonte completo do BRAMS (400.000 linhas de código), que contém 90% de código na linguagem *Fortran 90* e o restante em *C*. Contudo, devido a complexidade e estrutura do código disponibilizado pelo CPTEC/INPE, optou-se por trabalhar apenas com o código do módulo da dinâmica (110.000 linhas de código). Esse módulo foi separado do código total do BRAMS, o que possibilitou executá-lo com os mesmos dados de entrada fornecidos ao BRAMS. O código tem sido utilizado para fins de pesquisa, como descrito no Anexo B.

Para execução desse código são necessárias todas as ferramentas utilizadas para compilação e execução do BRAMS completo, que são: Mpich, Zlib, SZip, e Hdf5, compilador *Fortran* Gfortran ou Ifort. O ambiente de trabalho para execução do código da dinâmica foi instalado e configurado no HARP. Após a conclusão dessa tarefa, que consumiu tempo considerável, o

mentados 19 *kernels*. Essa foi uma das etapas mais trabalhosas e que consumiu muito tempo durante o desenvolvimento deste projeto. Alguns pontos dessa fase de seleção e codificação serão detalhados a seguir. Todos os códigos-fontes em *Fortran* que foram selecionados e codificados em *OpenCL* e que compõem a biblioteca de hardware são apresentados no Anexo A desta tese.

5.3 Framework para coprojeto hardware/software

Esse trabalho de doutorado teve como um de seus objetivos projetar, implementar e avaliar um *framework* para coprojeto de hardware/software do modelo climático BRAMS. A arquitetura do *framework* foi inspirada no Stretch. Optou-se por essa decisão em razão da arquitetura de computação deste processador incluir hardware especializado para execução de *kernels* previamente selecionados e hardware de propósito geral.

O Ambiente de Desenvolvimento Integrado (do inglês, *Integrated Development Environment* (IDE)) do Stretch unifica o desenvolvimento de software com instruções/blocos de hardware específicos (*Instruction Set Extension Fabric* (ISEF)) para computação intensiva para os processadores de software configuráveis Stretch. A IDE dispõe de um compilador, o *Stretch C Compiler* (SCC), *C/C++* que integra de forma completa no processo de compilação o código *C/C++* desenvolvido com as instruções de extensão em hardware (STRETCH, 2018). A Figura 26 ilustra a IDE do Stretch.

Figura 26 – Stretch - Ambiente de desenvolvimento de software

Fonte: Stretch (2018).

O SCC otimiza o código da aplicação e o conjunto de instruções do processador no qual a aplicação será executada. Além disso, o ambiente de desenvolvimento fornece ferramentas

para análise e depuração da aplicação. O SSC também dispõe de otimizações para a geração do hardware das instruções de extensão, possibilitando que os usuários possam usar um grande número das instruções disponíveis no ISEF. De acordo com [Stretch \(2018\)](#) gerar uma instrução de extensão para o processador é tão fácil quanto escrever e compilar uma função *C* e dispensa o conhecimento de linguagens de hardware ([STRETCH, 2018](#)).

Na arquitetura inicialmente proposta pretendia-se utilizar o *kit* DE2i-150 em virtude da proposta deste doutorado que previa também a execução do código do BRAMS em um ambiente heterogêneo composto por CPU e FPGA. Em seguida o projeto seria migrado para a placa BittWare S5PH-Q (durante a definição do projeto, o LCR ainda não dispunha dessa tecnologia, encontrava-se em processo de compra, por esse motivo o *kit* DE2i-150 foi a primeira opção). Por fim, estava a migração do projeto para a plataforma final de execução no ambiente do HARP. A Figura 28 ilustra o *framework* inicialmente proposto.

Figura 27 – Arquitetura inicial do projeto (Adaptada)

Fonte: [Stretch \(2018\)](#).

Com o andamento do projeto e a aprovação da proposta de projeto feita ao HARP pelos pesquisadores do LCR em conjunto com o INPE. Foram realizados alguns testes nas plataformas DE2i-150 e na BittWare, mas todo o desenvolvimento do sistema final foi concentrado na plataforma HARP. Além disso, a proposta inicial era utilizar todo o código do BRAMS. Contudo, em conversa com os desenvolvedores do BRAMS do INPE/CPTEC, optou-se por trabalhar apenas com o código da dinâmica, pois este código representa o futuro do padrão de código do BRAMS.

O código do BRAMS em produção no INPE/CPTEC engloba diferentes versões de *Fortran*, além de possuir dependências de dados entre os módulos que compõe o sistema. Essa dependência de dados, por exemplo, faz com que o BRAMS apresente problemas de balanceamento de carga. Uma possível solução para esse problema envolveria migrar processos e essa migração envolveria mover todos os dados desse processo para o novo processador. Como há vários módulos compondo o BRAMS, o esforço para implementar essa solução exigiria muito tempo e mão de obra que o CPTEC/INPE não dispõe no momento.

Dessa forma, como estudo de caso utilizado neste trabalho optou-se por usar o módulo de dinâmica do sistema BRAMS (CPTEC/INPE, 2018), mais informações sobre esse código está disponível no Anexo B. A Figura 28 ilustra a arquitetura do *framework* desenvolvido.

Figura 28 – Diagrama da arquitetura do *framework*

Fonte: Elaborada pelo autor.

5.3.1 Conversão de código Fortran para OpenCL

Todos os códigos selecionados são escritos em *Fortran 90*. Nessa fase não foi utilizada nenhuma ferramenta para conversão automática de *Fortran* para *OpenCL* ou *Fortran* para *C*, uma vez que a linguagem *C* possui sintaxe e semântica muito próxima do *OpenCL*. Não foi encontrada nenhuma ferramenta que automatizasse o processo de conversão de *Fortran* para *OpenCL*, além disso não era desejável inserir mais *overhead* no código, além do resultante do compilador *OpenCL* para a geração do hardware. O mesmo se aplica às ferramentas que poderiam automatizar a conversão de *Fortran* para *C*. Também não foi encontrada uma ferramenta que fizesse tal conversão para o código do BRAMS.

Outros fatores que influenciaram na escolha pela conversão de forma manual são a utilização de índices fixos de acesso a valores nas matrizes descritas no *Fortran*, presentes no código do BRAMS. As matrizes utilizadas pelo código do BRAMS, em boa parte, são tri-dimensionais e os valores contidos nas matrizes em *Fortran* são armazenados com orientação

a coluna, diferente da linguagem *C*, que é orientado a linha, o mesmo acontece para o *OpenCL*. Além dos fatores citados, o que também influenciou na escolha pela conversão manual foi a verificação dos resultados produzidos pelos códigos convertidos.

Nesta etapa foi criado um mecanismo de validação, que fornece os mesmos dados de entrada extraídos das entradas do BRAMS para execução dos códigos convertidos. Ao final da execução do código *Fortran*, selecionado da dinâmica do BRAMS (de acordo com a Figura 25), e o código *C* correspondente, as saídas foram comparadas. Nesse processo, a origem de divergências de valores era apontada, caso acontecessem. Esse processo foi trabalhoso em virtude do conhecimento reduzido da linguagem *Fortran* pelo pesquisador. Outra dificuldade foi o mapeamento entre os valores das matrizes nas duas linguagens que diferiam na forma de acesso às matrizes. A forma de acesso aos valores nas matrizes em *Fortran* era distinta da forma de acesso em *C*.

Dado que o código *OpenCL* assemelha-se ao código *C*, uma vez validado o código *C* e convertido a partir do código *Fortran*, o processo de criação do *kernel* em *OpenCL* não consumia muito tempo, pois era necessário adicionar as palavras chave da linguagem *OpenCL* e fazer a verificação e síntese usando o compilador *OpenCL*.

A Figura 29 ilustra o processo para gerar um código *OpenCL* correspondente a um código *Fortran*. O código *Fortran* possui os dados de entrada para serem executados e uma chamada para o seu código em *C* correspondente fornecendo os mesmos dados. Os códigos são compilados usando seus respectivos compiladores (*gfortran* ou *ifort*, *gcc* ou *icc*), para a geração dos *.o* de cada código. Em seguida, esses arquivos formam um executável (*.exe*), que é processado. Em caso da saída ser igual, a codificação do *OpenCL* pode ser realizada, se a saída divergir o processo é reiniciado buscando corrigir o erro identificado.

Os 19 *kernels* selecionados foram implementados usando os dois modelos de programação disponíveis para o Intel *OpenCL*. Inicialmente, todos foram programados no modelo *Task*, em seguida no modelo *NDRange*. No Capítulo 6 são apresentados os resultados obtidos para cada um dos *kernels* para cada modelo de programação.

É importante destacar que o Intel *OpenCL* dispõe de diferentes parâmetros de compilação para geração do hardware a partir de um código *OpenCL*. Além dos diferentes parâmetros, é possível variar o número de *work-items* que impacta diretamente no número de *work-groups* a ser gerado, o número de unidades de computação, dentre outros fatores. Considerando o elevado tempo de compilação para cada uma das alterações possíveis e a disponibilidade de uma máquina capaz de compilar o *OpenCL* e gerar hardware, restringiu-se a avaliar algumas variações no código *OpenCL*.

Figura 29 – Descrição do fluxo de conversão de código *Fortran* para *OpenCL*

Fonte: Elaborada pelo autor.

5.3.2 Compilação do kernel OpenCL

A compilação do *kernel OpenCL* foi também uma etapas onerosas com relação a tempo e esforço. A Figura 30 ilustra o processo de compilação de um arquivo *OpenCL* (.cl) para geração do seu hardware correspondente, o arquivo .aocx. Simular um hardware em *OpenCL* é uma tarefa rápida, no entanto, pode acontecer do hardware gerado não corresponder ao hardware simulado.

No contexto deste trabalho, o processo de geração de um hardware em *OpenCL* consumiu em média **8 horas de compilação**. Em alguns casos, mesmo decorrido esse tempo o hardware pode não ser gerado. Uma das causas para isso é a falta de memória RAM durante o processo de compilação. Para esse processo de geração de hardware uma exigência mínima é que a máquina possua pelo menos 40Gb de memória RAM. O ambiente de computação disponível no HARP é restrito a execução. Apenas o código do host é compilado no ambiente HARP. O LCR (Laboratório de Computação Reconfigurável do ICMC/USP) dispõe de uma máquina compartilhada entre os alunos do grupo, capaz de gerar hardware para o FPGA Arria 10 presente no HARP.

No escopo deste trabalho, esse foi o motivo para a maior parte dos casos em que o

hardware não foi gerado. Em outros casos, a não geração do hardware deveu-se a estimativas do início da compilação compreendendo recursos lógicos do FPGA entre 80 a 90% do hardware. O problema em ambos os casos é que o erro só ocorria quase ao final da compilação, ou seja, após 7.5 a 8 horas do início, problema que acabou consumindo muito tempo.

Figura 30 – Fluxo de compilação de um código *OpenCL*

Fonte: Intel (2018b).

5.3.3 Execução dos kernels no HARP

Uma vez gerado o hardware correspondente a um *kernel* é necessário codificar o código host que irá utilizar esse *kernel*. A codificação é feita na linguagem de programação C/C++. Se a construção do *kernel* não consome muito tempo, construir o código host para um *kernel* é um processo demorado. Ele envolve identificar o dispositivo, criar os buffers de memória, mapeá-los e inicializá-los, fornecer os argumentos, disparar o comando de execução do *kernel*, pegar o resultado do processamento e liberar os recursos do dispositivo que foram alocados para a execução do *kernel*.

A efeito de comparação, para um *kernel* com 20 linhas de código seu host possuirá em média 200 a 300 linhas. Após codificar os hosts para cada um dos *kernels* eles foram testados de forma *stand-alone*, ou seja, não foram integrados junto com o código da dinâmica do BRAMS. A ideia era executar os *kernels* na plataforma HARP com o objetivo de identificar erros durante a execução e desenvolver um método que permitisse fazer a integração de forma automática, de

acordo com o *kernel* selecionado para ser executado. Posteriormente, os *kernels* foram acoplados ao BRAMS.

5.3.4 Integração dos kernels ao código Fortran da dinâmica do BRAMS

Essa foi a fase mais onerosa do desenvolvimento do trabalho. Ela incluiu integrar as bibliotecas do Intel *OpenCL* com as bibliotecas do código da dinâmica do BRAMS, conectar o código do host com o código da dinâmica do BRAMS, gerar um executável e após a execução deste verificar se a saída era a esperada. O processo de integração dos *kernels* com o código em *Fortran* da dinâmica do BRAMS é ilustrado na Figura 31. *Fortran* e *OpenCL* não “conversam” de forma direta. Porém, como o host que faz a interface da aplicação em software com o *kernel* em hardware é escrito em *C/C++* foi possível utilizar da interoperabilidade que tanto *C/C++* possui com *Fortran*. Os códigos são compilados de forma separada e depois são integrados, formando um executável com as chamadas necessárias para que o código *Fortran* da dinâmica possa fazer uso do *kernel* em *OpenCL* programado no FPGA.

Figura 31 – Fluxo de integração e execução de um *kernel* com a dinâmica do BRAMS

Fonte: Elaborada pelo autor.

Com integrado o *kernel OpenCL* com *Fortran* a próxima fase é a execução do código totalmente integrado fazendo uso da CPU e do FPGA. Aqui, alguns problemas surgiram com alguns *kernels*, pois a execução destes provocava o estouro da condição de contorno para manter a simulação do modelo estável. O código do BRAMS nesse aspecto é muito bem escrito e

documentado, pois uma vez ultrapassado o valor da condição de contorno, que é calculada no início de cada *timestep* a execução do modelo é abortada pelo próprio sistema, informando onde ocorreu o erro de execução. Após experimentos e conversas com alguns desenvolvedores do BRAMS, foi possível identificar o problema. A identificação foi realizada de duas formas, a primeira consistiu em retirar a chamada de execução do hardware para o trecho de código substituído e em seu lugar colocar o código *C* que serviu de base para a geração do *kernel* em *OpenCL* e executá-lo juntamente com o código da dinâmica.

Inicialmente, a abordagem pretendia verificar se o código *C* que deu origem ao *kernel* em *OpenCL* estava correto ou se o problema tinha como origem o código em *C*. A segunda abordagem consistiu em imprimir os valores do cálculo da condição de contorno antes do *timestep* em que a execução do modelo era abortada quando havia a presença do hardware e comparar esses valores com os valores gerados com a execução do modelo sem a presença do hardware. Ambas as abordagens ajudaram a identificar o problema que causava a instabilidade do modelo e fazia com que a execução do mesmo fosse abortada.

No processo de criação do *kernel*, a simulação em *OpenCL* é rápida de ser realizada, no entanto, nem sempre os resultados finais correspondem ao hardware gerado, em alguns casos, durante a simulação do hardware em conjunto com o código da dinâmica do BRAMS a condição de contorno não estourava. Quando o hardware era programado no FPGA o comportamento era diferente e, nesses casos específicos, eram erros no posicionamento de alguns mecanismos do *OpenCL* na execução concorrente dos *work-items*, como por exemplo, o uso de barreiras no acesso à memória.

5.3.5 Instrumentação do código da dinâmica para invocar o kernel em OpenCL

Um dos objetivos definidos para este doutorado era instrumentar os trechos de código em *Fortran*. Após selecionados para serem codificados em hardware, no momento da sua chamada de execução, acontecia a indicação para este ser executado no FPGA e não na CPU.

Essa etapa consistiu na criação de modelos de código para cada um dos *kernels* com o objetivo de automatizar o processo de instrumentação e na execução dos testes realizados no HARP. O acesso ao HARP é realizado de forma remota, usando ssh, por exemplo. Considerando o tempo de acesso ao HARP, compilação e integração do host com o código da dinâmica e a disponibilidade do recurso no momento da execução do sistema, um teste consumia em média 10 a 15 minutos.

Uma ferramenta muito útil para auxiliar em todo esse processo foi a IntelliJ IDEA ¹. IntelliJ IDEA é uma IDE proprietária, que fornece licença gratuita de 1 ano para estudantes. Ela é uma poderosa ferramenta para desenvolvimento de software, pois ela integra acesso

¹ <https://www.jetbrains.com/idea/>

remoto e disponibiliza uma interface gráfica de programação com vários plugins que auxiliam o programador. Essas facilidades possibilitaram alterar, compilar e executar o sistema de forma mais rápida do que usar o terminal linux em conjunto com outras ferramentas em separado.

A Figura 61 ilustra o processo de instrumentação do código da dinâmica fazendo uso da CPU e do FPGA para seus respectivos trechos de código. Foi construída uma base de modelos para os *kernels* implementados em *OpenCL*. No momento da compilação, uma vez escolhido o *kernel* a ser executado, o módulo de instrumentação realiza buscas na base de modelos e substitui no código da dinâmica o código correspondente a invocação do código *Fortran* que se liga ao restante do "software". Ele é responsável por invocar o hardware correspondente ao *kernel*. Uma vez escolhido o *kernel* a ser executado, a biblioteca de hardware é consultada e é disparado o comando para programar no FPGA o arquivo *.aocx* correspondente ao *kernel*, pois no momento da execução este deve receber os dados a serem processados. Por fim, todo o restante do "software" é executado na CPU.

Figura 32 – Diagrama de blocos do módulo de instrumentação do código da dinâmica

Fonte: Elaborada pelo autor.

Com esse módulo pretende-se numa próxima versão do *framework* integrar de forma automatizada as etapas de geração e análise do profile e a construção do hardware em *OpenCL*, alimentando dessa forma a biblioteca de modelos e a de hardware para os módulos restantes do BRAMS. Devido ao esforço despendido em entender as estruturas do código do BRAMS será possível construir um parser que dado um código *Fortran* o converta para *OpenCL*, integrando

também o método de verificação de compatibilidade dos códigos gerados que já faz parte do *framework*.

5.3.6 Framework

A Figura 33 ilustra uma tela do *framework* desenvolvido durante o doutorado. A interface gráfica foi desenvolvida com o objetivo de facilitar a usabilidade com o usuário e facilitar a utilização do *framework*. Nesta tela o usuário tem acesso às funcionalidades do *framework* para selecionar os parâmetros de execução e acompanhar a execução da dinâmica do BRAMS.

Figura 33 – Tela principal do *framework* proposto

Seleção dos Parâmetros de Execução:

Selecione o FPGA:

Arria 10
Stratix V

Selecione o Módulo:

Turbulência

Selecione a Resolução:

60X60

Selecione o Kernel:

Kernel 1
Kernel 2
Kernel 3
Kernel 4

Descrição do Kernel Selecionado:

Kernel 1 referente a sub-rotina diffsc/r para o cálculo da turbulência por difusão. A sua chamada encontra-se no arquivo turb_diff.f90.

Execução da Previsão do Tempo

```

BRAMS 5 Version 18 on INITIAL run
! input namelist filename is RAMSIN
! parallel execution with 1 processes

=== Finish initialization; Wall(sec)=0.31

Timestep 1: Sim Time 60.0s; Wall 0.191s
Timestep 2: Sim Time 120.0s; Wall 0.191s
Timestep 3: Sim Time 180.0s; Wall 0.191s
Timestep 4: Sim Time 240.0s; Wall 0.191s
Timestep 5: Sim Time 300.0s; Wall 0.191s
Timestep 6: Sim Time 360.0s; Wall 0.191s
Timestep 7: Sim Time 420.0s; Wall 0.195s
Timestep 8: Sim Time 480.0s; Wall 0.191s
Timestep 9: Sim Time 540.0s; Wall 0.191s
Timestep 10: Sim Time 600.0s; Wall 0.191s
Timestep 11: Sim Time 660.0s; Wall 0.191s
Timestep 12: Sim Time 720.0s; Wall 0.191s
Timestep 13: Sim Time 780.0s; Wall 0.199s
Timestep 14: Sim Time 840.0s; Wall 0.191s
Timestep 15: Sim Time 900.0s; Wall 0.191s
Timestep 16: Sim Time 960.0s; Wall 0.191s
Timestep 17: Sim Time 1020.0s; Wall 0.191s
Timestep 18: Sim Time 1080.0s; Wall 0.191s
Timestep 19: Sim Time 1140.0s; Wall 0.188s
Timestep 20: Sim Time 1200.0s; Wall 0.195s
Timestep 21: Sim Time 1260.0s; Wall 0.191s
Timestep 22: Sim Time 1320.0s; Wall 0.191s
Timestep 23: Sim Time 1380.0s; Wall 0.191s
Timestep 24: Sim Time 1440.0s; Wall 0.191s
Timestep 25: Sim Time 1500.0s; Wall 0.188s
Timestep 26: Sim Time 1560.0s; Wall 0.195s
Timestep 27: Sim Time 1620.0s; Wall 0.191s
Timestep 28: Sim Time 1680.0s; Wall 0.191s
Timestep 29: Sim Time 1740.0s; Wall 0.191s
Timestep 30: Sim Time 1800.0s; Wall 0.191s
Timestep 31: Sim Time 1860.0s; Wall 0.191s
Timestep 32: Sim Time 1920.0s; Wall 0.191s
Timestep 33: Sim Time 1980.0s; Wall 0.195s
Timestep 34: Sim Time 2040.0s; Wall 0.191s
  
```

Executar **Sair**

Fonte: Elaborada pelo autor.

Para iniciar uma sessão o usuário seleciona o FPGA, o módulo e a resolução que deseja executar a dinâmica do BRAMS. Em seguida escolhe o *kernel* para o qual o *framework* apresenta uma breve descrição para auxiliar o usuário. Com a configuração desejada o usuário acompanha a execução do módulo da dinâmica pela interface gráfica.

RESULTADOS

6.1 Considerações iniciais

Este capítulo relata os experimentos realizados com o objetivo de avaliar a aplicabilidade do *framework* proposto nesta tese de doutorado.

O capítulo está organizado da seguinte forma. Na Seção 6.2 são apresentadas as fases iniciais dos estudos de caso, como a preparação e a execução. Também são apresentados os resultados e discussão.

6.2 Estudos de caso: preparação e execução

Para a condução dos estudos de caso foi utilizada a plataforma do projeto HARP, cuja estrutura de computação é um cluster.

No caso do HARP, foi utilizado um nó do cluster que contém um processador Intel Xeon de 2.4 GHz e um FPGA Arria 10 integrado, Sistema Operacional Linux Ubuntu versão 16.04.

O delineamento dos estudos de caso considerou diferentes implementações e plataformas de execução. No estudo de caso I os *kernels* foram implementados usando os modelos de programação *Task* e *NDRange* do Intel *OpenCL* e foram executados no FPGA Arria 10. No estudo de caso II os *kernels* foram implementados usando o modelo de programação *NDRange* do Intel *OpenCL* e também foram executados no FPGA Arria 10. No estudo de caso III os *kernels* implementados usando o modelo de programação *NDRange* do estudo de caso II foram executados no FPGA Stratix V. As próximas seções detalham os estudos e resultados obtidos.

Os códigos fontes dos *kernels* que foram desenvolvidos em *OpenCL* neste trabalho de doutorado são apresentados no Anexo A e no Apêndice A é apresentada uma visão do sistema dos *kernels*.

6.3 Estudo de caso I

Para este estudo foram implementados 19 *kernels*. Para cada um dos *kernels* foram fornecidas implementações considerando os modelos de programação (*Task* e *NDRange*) do Intel *OpenCL*, cópia de dados para memória interna do FPGA, acesso aos dados na memória global e variação quanto ao número de *Work-Items* (WI) e *Work-Groups* (WG). Os *kernels* implementados foram divididos em 4 grupos considerando o módulo da dinâmica, são eles: *turbulência*, *grad*, *advect* e *acustic*. Além das características do hardware, o código da dinâmica foi executado para duas resoluções de grade 35X60X60 e 35X100X100. Nas próximas subseções são apresentados os resultados para cada um dos grupos.

6.3.1 Kernels da turbulência implementados em Task

Os dados apresentados na Tabela 1 correspondem as médias dos tempos de execução para cada um dos *kernels* em conjunto com o código da dinâmica do BRAMS. As colunas (T sw(us)) e três (T hw(us)) apresentam os dados de tempo dos *kernels* em software e em hardware em microsegundos, respectivamente. Os valores representados nestas colunas foram obtidos considerando o trecho de código referente ao *kernel* em software e ao *kernel* em hardware. As colunas quatro (T F-C(us)) e cinco (T C-F(us)) apresentam os valores dos tempos de cópia dos dados para serem processados e o resultado do processamento. Na quarta coluna está o tempo quando os dados são transferidos do código *Fortran* para o código do host em C a ser enviado para o *kernel* no FPGA. Na quinta coluna está o tempo quando os dados são transferidos do código C para o código *Fortran*. A Tabela 1 também representa os recursos de hardware do FPGA que foram utilizados por cada um dos *kernels*, assim como a frequência máxima obtida em cada um deles. A coluna TED(s) corresponde ao tempo da execução completa do código da dinâmica do BRAMS.

Apesar dos *kernels* 1 e 3 terem alcançado aproximadamente a mesma frequência, a diferença no tempo de execução está relacionada com a quantidade de vezes que cada um dos *kernels* é chamado. O *kernel* 1, por exemplo, é chamado 25920 vezes, enquanto que o *kernel* 2 é chamado 4320 vezes.

Essa foi a primeira implementação codificada, compilada e executada, pois buscava avaliar se o modelo de programação do host usando memória virtual compartilhada dispensava a cópia dos dados a serem processados pelo *kernel* para a memória interna do FPGA. Memória virtual compartilhada foi um recurso adicionado ao *OpenCL* na versão 2.0. A operação de cópia de dados para memória local do FPGA é uma operação custosa e essa implementação buscava reduzir esse custo de cópia dos dados.

Para este estudo de caso uma execução do código da dinâmica sem *kernels* em hardware correspondeu a um tempo médio de 108.6 segundos. Ao observar os dados apresentados na Tabela 1 a execução global do sistema foi aumentada no pior caso em 800 vezes (*kernel* 1) e no

melhor caso em 4 vezes (*kernel* 2).

Tabela 1 – *Kernels* da turbulência sem cópia de dados para memória local para o modelo Task de programação

K	T sw(us)	T hw(us)	T F-C(us)	T C-F(us)	EL(%)	R(%)	F(MHz)	B(MB/s)	TED(s)
1	427.9873	330609.4567	589.9066	103.9427	28	27	234.3	8	8694.6
2	129.6997	8924.3765	777.1000	108.8876	26	24	215.6	338.93	421.9
3	495.9106	175380.1044	512.8349	189.0150	26	21	234.4	14	887.7
4	762.9395	370024.8130	576.5648	266.3496	31	29	218.8	8	1721.2

Nota – k - *kernel*, T sw - tempo do *kernel* em software, T hw - tempo do *kernel* em hardware, T F-C - tempo de cópia *Fortran* para C, T C-F - tempo de cópia do C para o *Fortran*, EL - elementos lógicos, R - RAM *blocks*, F - frequência máxima, B - *bandwidth*, TED = tempo de execução da dinâmica

Esse primeiro estudo de caso permitiu fornecer uma nova implementação para esse mesmo grupo de *kernels*, considerando cópia dos dados da memória global (off-chip) do FPGA para a memória local (on-chip). Essa decisão também considerou que o FPGA Arria 10 utilizado dispõe de aproximadamente 7 MB de memória on-chip. Cada matriz a ser armazenada ocupa um espaço de aproximadamente 0.5MB. Os dados dessa segunda versão são apresentados na Tabela 2.

Nesta Tabela, observa-se uma significativa melhora em relação ao pior caso de tempo do hardware da Tabela 1. No entanto, essa melhora não é suficiente para igualar ou superar o tempo de execução da dinâmica utilizando apenas a CPU.

Tabela 2 – *Kernels* da turbulência com cópia de dados para memória local no modelo Task de programação

K	T sw(us)	T hw(us)	T F-C(us)	T C-F(us)	EL(%)	R(%)	F(MHz)	B(MB/s)	TED(s)
1	427.9873	79217.2486	729.8999	113.5782	28	66	175.92	32.00	2200.2
2	129.6997	4717.9717	1049.2520	127.9745	26	74	209.2	641.55	327.6
3	495.9106	21784.0885	657.9067	203.5983	27	64	172.5	115.69	251.8
4	762.9395	52221.7628	699.8027	282.4807	30	75	167.5	58.02	368.5

Nota – k - *kernel*, T sw - tempo do *kernel* em software, T hw - tempo do *kernel* em hardware, T F-C - tempo de cópia *Fortran* para C, T C-F - tempo de cópia do C para o *Fortran*, EL - elementos lógicos, R - RAM *blocks*, F - frequência máxima, B - *bandwidth*, TED = tempo de execução da dinâmica

A Tabela 2 também apresenta os recursos de hardware para os *kernels* da turbulência que fazem cópia dos dados da memória global do FPGA para a memória local. Uma grande diferença a ser observada nessa implementação é o aumento no uso dos blocos de memória. Comparando com a primeira versão apresentada na Tabela 1 o uso de memória dobrou. Outra característica que pode ser observada é a redução da frequência do hardware gerado, que nesse caso está diretamente ligada ao aumento do uso de mais blocos de memória, pois a quantidade de elementos lógicos utilizados praticamente não se alterou. Apesar de possuir uma frequência mais baixa em relação as versões dos *kernels* que não fazem cópia dos dados para a memória local, os *kernels* que fizeram cópia para a memória local apresentaram melhor desempenho pelo fato de

obterem mais rapidamente os dados a serem processados. Dessa forma, foi evitado o overhead para buscar, processar e escrever o dado na memória global. Os *kernels* que fazem cópia dos dados da memória global para a memória local só escrevem os dados de volta na memória global ao final do processamento.

Sumarizando os resultados para o módulo da turbulência seguindo o modelo Task, a Figura 34 apresenta uma visão geral dos tempos para a execução dinâmica dos *kernels*.

Figura 34 – Execução dinâmica dos *kernels* do módulo da turbulência para o modelo Task

Fonte: Elaborada pelo autor.

6.3.2 Kernels da turbulência implementados em NDRange

Com a experiência adquirida da programação em Task e buscando melhorar os tempos de execução dos *kernels* foi feita uma nova implementação dos *kernels* da turbulência usando o modelo de programação do Intel *OpenCL* em NDRange. Nestes estudos de caso procurou-se variar o número de *work-items* que consequentemente impacta no número de work-groups gerados e no número de unidades de computação. Este número informa ao compilador *OpenCL* quantas instâncias do hardware devem ser replicadas. No primeiro estudo foi criado um hardware com 20 *work-items* por dimensão e apenas 1 unidade de computação. Os resultados dessa implementação são apresentados na Tabela 3.

Diferente da implementação em Task para os *kernels* da turbulência a implementação em NDRange comprovou ser mais eficiente e produziu um resultado superior. Ainda de acordo com a Tabela 3 é possível observar que o hardware gerado em relação a frequência máxima é melhor do que o gerado pela programação em Task. Outro fator que contribui para a implementação

Tabela 3 – *Kernels* da turbulência para o modelo NDRange para 1 CU e WI(20,20) na resolução 35X60X60

K	T sw(us)	T hw(us)	T F-C(us)	T C-F(us)	EL(%)	R(%)	F(MHz)	B(MB/s)	TED(s)
1	427.9873	7969.9910	659.4290	78.0660	27	20	264.76	325	369.4
2	129.6997	5761.8357	573.4720	109.5420	26	18	228.78	515	343.8
3	495.9106	8160.9595	611.2430	205.0370	26	17	240.00	310	195.6
4	762.9395	9858.9318	764.9660	282.1060	30	23	218.67	310	195.5

Nota – CU - *Compute Unit*, K - *kernels*, T sw - tempo do *kernel* em software, T hw - tempo do *kernel* em hardware, T F-C - tempo de cópia *Fortran* para C, T C-F - tempo de cópia do C para o *Fortran*, EL - elementos lógicos, R - RAM *blocks*, F - frequência máxima, B - *bandwidth*, TED = tempo de execução da dinâmica

do hardware em NDRange ter produzido um melhor resultado é justificado por apresentar semelhança com o modelo de programação e execução do BRAMS que usa MPI para fazer a divisão dos dados entre os processos durante a execução. Essa mesma divisão proporcionada pelo MPI ocorre na programação NDRange por meio dos *work-items*. Por exemplo, se um hardware possuir 20 *work-items* por dimensão significa que existem 400 threads processando em paralelo o trecho de código convertido em hardware. Porém, no caso do BRAMS existe um problema nessa configuração, pois o código do BRAMS é paralelo apenas nas dimensões x e y, o que não ocorre na dimensão z, onde ocorre todo o cálculo ou processamento do modelo.

Com os resultados obtidos com a primeira implementação dos *kernels* da turbulência em NDRange foi gerada uma segunda implementação desse código variando o número de *work-items* de 20 para 30 por dimensão e o número de unidades de computação de 1 para 2. A especificação do número de *work-items* considerou a resolução 35(z)X60(x)X60(y), em que o módulo da dinâmica é executado. Também foram utilizadas apenas 2 dimensões 20X20 ou 30X30 em termos de *work-items*, pois apenas as dimensões x e y da resolução são paralelas, a dimensão z é executada sequencialmente. Os dados são apresentados na Tabela 4. Esperava-se que ao aumentar o número de *work-items* e unidades de computação o tempo de execução final da dinâmica fosse reduzir, o que não ocorreu. Além disso, a frequência do hardware gerado foi reduzida na maioria dos casos e ocorreu uma degradação no tempo de execução. Isto pode ser explicado em partes por uma competição dos *work-groups* para acessar a unidade de computação e a memória. Outro indício é que com o aumento no número de *work-items* aumenta-se o tempo de sincronismo entre eles na produção do resultado final.

Os dados apresentados na Tabela 5 referem-se a terceira versão em NDRange dos *kernels* da turbulência. Nessa implementação foi reduzido para 20X20 o número de *work-items* por dimensão e a quantidade de unidades de computação foi aumentada para 10 para os *kernels* 1, 2 e 3 e para 6 para o *kernel* 4. O número de unidades de computação não se manteve o mesmo para todos os *kernels* em virtude do tamanho do hardware gerado. O *kernel* 4 foi o único que não possibilitou gerar com 10 unidades de computação, pois excedia o número de recursos disponíveis no FPGA.

Essa implementação produziu uma significativa melhora no tempo de execução da dinâ-

Tabela 4 – *Kernels* da turbulência para o modelo NDRange para 2 CU e WI(30,30) na resolução 35X60X60

K	T sw(us)	T hw(us)	T F-C(us)	T C-F(us)	EL(%)	R(%)	F(MHz)	B(MB/s)	TED(s)
1	427.9873	8574.9596	652.8210	94.2470	31	26	227.50	290	386.8
2	129.6997	5871.8626	642.6140	112.8030	27	21	216.60	510	348.6
3	495.9106	8360.5747	642.2510	217.5970	28	20	242.80	310	195.8
4	762.9395	10844.5985	751.5010	333.4960	36	31	203.58	310	200.0

Nota – CU - *Compute Unit*, K - *kernels*, T sw - tempo do *kernel* em software, T hw - tempo do *kernel* em hardware, T F-C - tempo de cópia *Fortran* para C, T C-F - tempo de cópia do C para o *Fortran*, EL - elementos lógicos, R - RAM *blocks*, F - frequência máxima, B - *bandwidth*, TED = tempo de execução da dinâmica

Tabela 5 – *Kernels* da turbulência para o modelo NDRange para 10 CU e WI(20,20) na resolução 35X60X60

K	T sw(us)	T hw(us)	T F-C(us)	T C-F(us)	EL(%)	R(%)	F(MHz)	B(MB/s)	TED(s)
1	427.9873	5819.0903	475.9459	54.3781	53	70	172.50	434	251.9
2	129.6997	3287.5993	537.3587	58.5910	39	49	212.50	922	206.7
3	495.9106	3674.4287	508.1359	115.3353	42	42	197.90	685	123.3
4	762.9395	9329.6983	640.9076	157.6595	56	64	160.00	326	149.2

Nota – CU - *Compute Unit*, K - *kernels*, T sw - tempo do *kernel* em software, T hw - tempo do *kernel* em hardware, T F-C - tempo de cópia *Fortran* para C, T C-F - tempo de cópia do C para o *Fortran*, EL - elementos lógicos, R - RAM *blocks*, F - frequência máxima, B - *bandwidth*, TED = tempo de execução da dinâmica

mica. Houve redução da frequência do hardware gerado, no entanto, a *bandwidth* praticamente dobrou para todos os *kernels*. Isso ajuda a explicar a melhora obtida, pois com um número maior de dados disponíveis para processamento reduz-se o tempo de espera do hardware entre buscar o dado na memória e processá-lo.

Por fim, a última variação feita nos *kernels* da turbulência foi produzir uma versão de hardware para executar o código da dinâmica com uma resolução de 35X100X100. O objetivo dessa implementação e teste foi verificar se o comportamento do hardware seria melhor ou pior dado uma quantidade de dados maior para ser processada em relação a resolução de 35X60X60. Os resultados dessa implementação são apresentados na Tabela 6. Vale destacar que ao alterar a resolução para executar a dinâmica do BRAMS esta não foi executada para uma versão do hardware no formato Task para a nova resolução de 35X100X100. O modelo Task de programação só produziu bons resultados quando os dados foram copiados para a memória local do FPGA. Desse modo, como a resolução do modelo foi aumentada, as matrizes que antes ocupavam 0.5MB passaram a ocupar 1.4MB na memória, não sendo possível gerar um hardware que copiasse todas as matrizes necessárias para a memória local.

A execução da dinâmica na resolução de 35X100X100 consome um tempo médio de 320.5s executando apenas na CPU. Para executar a dinâmica nessa nova resolução foi gerada apenas uma versão de *kernels* da turbulência os quais são apresentados na Tabela 6. Para esse caso, em virtude do tempo de compilação e disponibilidade da máquina, que estava sendo utilizada

Tabela 6 – *Kernels* da turbulência para o modelo NDRange para 2 CU e WI(50,50) na resolução 35X100X100

K	T sw(us)	T hw(us)	T F-C(us)	T C-F(us)	EL(%)	R(%)	F(MHz)	B(MB/s)	TED(s)
1	1164.9484	22802.0587	1380.3346	294.6211	31	26	226.39	307.64	1019.63
2	396.0304	13127.8702	1447.3622	296.8088	27	21	228.99	639.96	855.77
3	1371.7995	16613.2480	1146.0963	479.8824	28	20	246.00	421.53	438.89
4	2157.9164	25008.4384	1505.3294	682.3004	36	31	220.00	335.98	468.11

Nota – CU - *Compute Unit*, K - *kernels*, T sw - tempo do *kernel* em software, T hw - tempo do *kernel* em hardware, T F-C - tempo de cópia *Fortran* para C, T C-F - tempo de cópia do C para o *Fortran*, EL - elementos lógicos, R - RAM *blocks*, F - frequência máxima, B - *bandwidth*, TED = tempo de execução da dinâmica

para realizar a compilação do código *OpenCL* para gerar o hardware, não foram gerados outros hardwares variando a quantidade de *work-items* e unidades de computação. Pode-se perceber com os dados apresentados que o comportamento de execução segue o mesmo padrão do hardware gerado para a resolução de 35X60X60, não houve ganho de desempenho se comparado com a execução apenas na CPU.

Para o módulo da turbulência seguindo o modelo NDRange a Figura 35 ilustra uma visão geral dos tempos obtidos para a execução dinâmica dos *kernels*.

Figura 35 – Execução dinâmica dos *kernels* do módulo da turbulência para o modelo NDRange

Fonte: Elaborada pelo autor.

6.3.3 Kernels *grad* implementados em Task

Outro grupo de *kernels* implementados foi denominado de *grad*. Seguindo a mesma ideia utilizada com o grupo da turbulência foram feitas duas implementações em Task e duas implementações em NDRange para o grupo de *kernels grad*. Nesse grupo foram selecionados e implementados 5 *kernels*, que são apresentados na Tabela 7, a primeira implementação em Task.

Tabela 7 – *Kernels grad* sem cópia para memória local para o modelo de programação Task na resolução 35X60X60

K	T sw(us)	T hw(us)	T F-C(us)	T C-F(us)	EL(%)	R(%)	F(MHz)	B(MB/s)	TED(s)
1	270.6504	21818.0638	383.8761	122.5958	34	42	209.46	221.29	838.37
2	240.5633	4653.3863	308.5849	121.7061	34	42	206.52	1036.62	337.81
3	258.0949	21755.8781	366.7193	122.6256	34	42	213.49	221.54	833.51
4	260.9860	4698.5897	312.0719	121.4547	34	42	204.08	1029.19	337.50
5	43.5371	5952.6443	233.6768	108.0927	25	19	224.56	338.89	162.80

Nota – K - *kernel*, T sw - tempo do *kernel* em software, T hw - tempo do *kernel* em hardware, T F-C - tempo de cópia *Fortran* para C, T C-F - tempo de cópia do C para o *Fortran*, EL - elementos lógicos, R - RAM *blocks*, F - frequência máxima, B - *bandwidth*, TED = tempo de execução da dinâmica

Os dados apresentados correspondem a implementação em Task sem cópia de dados para a memória local do FPGA. Para esse grupo de *kernels* em específico, os *kernels* 2, 4 e 5 não copiar os dados para a memória local (ver Tabela 8 não teve um impacto significativo no tempo de execução ao analisar o que ocorreu com alguns *kernels* do grupo da turbulência. Outro ponto a ser destacado são os dados dos recursos de hardware utilizados pelos *kernels*. Os *kernels* 1, 2, 3 e 4 nesse aspecto são praticamente idênticos, no entanto, as operações que cada um deles realiza possuem uma ligeira diferença, como, por exemplo, um *kernel* pode realizar uma operação de soma, enquanto outro pode realizar uma subtração ou multiplicação. Os resultados apresentados para estes *kernels* correspondem a execução da dinâmica na resolução de 35X60X60.

A Tabela 8 apresenta a segunda versão da programação em Task para os *kernels* do grupo *grad*. Copiar os dados para a memória local para serem processados por esses *kernels* não ocasionou uma redução significativa no tempo total de execução. Além disso, em função dessa cópia de dados, a frequência do hardware foi reduzida para a maioria dos *kernels*. Houve uma variação de *bandwidth*, isto ajuda a explicar a redução no tempo de execução pela metade dos *kernels* 1 e 3. Também houve redução para os *kernels* 2, 4 e 5, no entanto, esta não foi tão expressiva.

Para o módulo *grad* seguindo o modelo Task a Figura 36 ilustra uma visão geral dos tempos obtidos para a execução dinâmica dos *kernels*.

Tabela 8 – kernels grad com cópia para memória local para o modelo de programação Task na resolução 35X60X60

K	T sw(us)	T hw(us)	T F-C(us)	T C-F(us)	EL(%)	R(%)	F(MHz)	B(MB/s)	TED(s)
1	270.6504	8340.5657	348.8081	126.8910	35	46	129.00	498.27	438.90
2	240.5633	4205.4339	363.6483	136.1663	34	44	139.00	984.36	321.07
3	258.0949	7319.0578	331.2701	126.0250	35	46	149.70	562.15	405.23
4	260.9860	4180.9051	353.1299	131.5903	35	46	150.00	983.15	319.67
5	43.5371	2944.8407	253.2061	113.7799	28	39	228.46	686.44	152.66

Nota – K - *kernel*, T sw - tempo do *kernel* em software, T hw - tempo do *kernel* em hardware, T F-C - tempo de cópia *Fortran* para C, T C-F - tempo de cópia do C para o *Fortran*, EL - elementos lógicos, R - RAM *blocks*, F - frequência máxima, B - *bandwidth*, TED = tempo de execução da dinâmica

Figura 36 – Execução dinâmica dos *kernels* do módulo grad para o modelo Task

Fonte: Elaborada pelo autor.

6.3.4 Kernels grad implementados em NDRange

Seguindo a metodologia de implementação e realização dos estudos de caso adotada para este trabalho serão apresentados os resultados das implementações em NDRange dos *kernels* do grupo grad. Destaca-se que foram implementados uma versão para cada resolução em que a dinâmica foi executada.

A Tabela 9 apresenta os dados dos *kernels* para execução da dinâmica na resolução 35X60X60. Todos os *kernels* foram gerados com 30 *work-items* nas dimensões x e y e 4 unidades de computação. O modelo de programação NDRange para esse grupo de *kernels* produziu um

hardware mais eficiente. No entanto, o mesmo não pode ser dito para o *kernel* 5, pois sua implementação em Task produziu um resultado melhor do que em NDRange. Em Task o seu tempo médio de execução foi de 152.66s, enquanto que em NDRange esse tempo foi de 193.29s. Mais uma vez os dados de recursos de hardware utilizados são similares para os *kernels* 1, 2, 3 e 4.

Tabela 9 – Kernels grad para o modelo de programação NDRange para 4 CU e WI(30,30) na resolução 35X60X60

K	T sw(us)	T hw(us)	T F-C(us)	T C-F(us)	EL(%)	R(%)	F(MHz)	B(MB/s)	TED(s)
1	270.6504	6306.3130	303.0553	63.3946	51	63	187.00	179.91	304.26
2	240.5633	2775.4457	232.6239	63.2877	51	63	202.00	407.85	195.07
3	258.0949	6104.2713	295.9178	64.3118	50	63	196.27	186.00	296.87
4	260.9860	2761.8714	232.0736	62.4981	50	63	208.63	409.13	193.01
5	43.5371	2751.3947	236.4900	63.3652	29	27	242.00	410.71	193.29

Nota – CU - *Compute Unit*, K - *kernels*, T sw - tempo do *kernel* em software, T hw - tempo do *kernel* em hardware, T F-C - tempo de cópia *Fortran* para C, T C-F - tempo de cópia do C para o *Fortran*, EL - elementos lógicos, R - RAM *blocks*, F - frequência máxima, B - *bandwidth*, TED = tempo de execução da dinâmica

Os dados apresentados na Tabela 10 referem-se a implementação dos *kernels* do grupo grad em NDRange para execução da dinâmica na resolução de 35X100X100. Esses dados seguem o mesmo comportamento que a versão dos *kernels* implementados para execução da dinâmica com a resolução 36x60x60. Vale destacar que o tempo médio de execução da dinâmica para a resolução de 35X100X100 é de 320.5s.

Tabela 10 – Kernels grad para o modelo de programação NDRange para 4 CU e WI(50,50) na resolução 35X100X100

K	T sw(us)	T hw(us)	T F-C(us)	T C-F(us)	EL(%)	R(%)	F(MHz)	B(MB/s)	TED(s)
1	740.2471	16205.6943	593.2278	179.3026	51	63	200.00	192.07	806.96
2	752.2762	8457.4723	447.3624	164.1700	51	63	194.00	367.74	584.76
3	739.8612	16724.5522	542.6564	181.7555	50	63	181.35	186.27	836.30
4	723.8358	8285.7637	448.3858	162.8983	50	63	199.96	375.69	576.55
5	154.3039	3639.1672	404.3317	169.7249	26	21	246.30	1538.85	334.50

Nota – CU - *Compute Unit*, K - *kernels*, T sw - tempo do *kernel* em software, T hw - tempo do *kernel* em hardware, T F-C - tempo de cópia *Fortran* para C, T C-F - tempo de cópia do C para o *Fortran*, EL - elementos lógicos, R - RAM *blocks*, F - frequência máxima, B - *bandwidth*, TED = tempo de execução da dinâmica

Para o módulo grad seguindo o modelo NDRange a Figura 37 ilustra uma visão geral dos tempos obtidos para a execução dinâmica dos *kernels*.

6.3.5 Kernels *advect* implementados em Task

A Tabela 11 refere-se aos dados obtidos da execução dos hardwares implementados no modelo Task para o grupo de *kernels* denominado *advect*. Os dados da implementação

Figura 37 – Execução dinâmica dos *kernels* do módulo grad para o modelo NDRange

correspondem aos *kernels* que buscam os dados de execução na memória global do FPGA. Também para este grupo de *kernels* os resultados obtidos do tempo de execução da dinâmica não foram satisfatórios. O *kernel 5* é o que apresenta o melhor resultado, no entanto, não iguala nem supera o tempo de execução da dinâmica usando apenas a CPU, que é de 107.12s para a resolução de 35X60X60.

Tabela 11 – *Kernels* advection para o modelo de programação Task sem cópia para memória local na resolução 35X60X60

K	T sw(us)	T hw(us)	T F-C(us)	T C-F(us)	EL(%)	R(%)	F(MHz)	B(MB/s)	TED(s)
1	315.9886	25210.2399	825.5839	204.9383	30	38	189.68	140.04	832.60
2	335.3631	23526.4799	792.4000	198.3553	30	38	225.52	149.99	786.04
3	354.5489	26942.2997	698.8717	103.6699	30	31	225.27	112.28	869.00
4	36.0993	8561.0337	375.9816	113.2940	25	19	228.00	176.64	408.94
5	586.0613	11384.9390	722.1070	755.4335	39	44	202.92	591.09	144.14

Nota – K - *kernel*, T sw - tempo do *kernel* em software, T hw - tempo do *kernel* em hardware, T F-C - tempo de cópia *Fortran* para C, T C-F - tempo de cópia do C para o *Fortran*, EL - elementos lógicos, R - RAM *blocks*, F - frequência máxima, B - *bandwidth*, TED = tempo de execução da dinâmica

Também foi fornecido uma implementação que copia os dados da memória global para a memória local do FPGA para os *kernels* do grupo advection. Os dados são apresentados na Tabela 12. Assim como ocorreu redução do tempo de execução da dinâmica com os *kernels* dos grupos turbulência e grad que fizeram cópia dos dados para a memória local, o mesmo ocorreu para os *kernels* do grupo advection em relação a implementação desses *kernels* que não fazem cópia

dos dados para a memória local. É interessante observar que nessa implementação era esperado que o número de uso de blocos de memória aumentassem, porém, houve redução no número de elementos lógicos utilizados e também da frequência obtida pelos hardwares gerados. Além disso, a largura de banda ao acessar a memória aumentou para os *kernels* 1, 2, 3 e 4, a exceção foi o *kernel* 5, que teve a largura de banda reduzida e um aumento no tempo de execução da dinâmica. Se na implementação que não fez cópia de dados para a memória local o *kernel* 5 obteve melhor desempenho, na implementação que fez cópia dos dados para a memória local ao avaliar a evolução do hardware ele (*kernel* 5) obteve o pior desempenho, vale destacar que o número de vezes que o (*kernel* 5) é chamado é cerca de 25x menos que os outros *kernels*.

Tabela 12 – *Kernels* advection para o modelo de programação Task com cópia para memória local na resolução 35X60X60

K	T sw(us)	T hw(us)	T F-C(us)	T C-F(us)	EL(%)	R(%)	F(MHz)	B(MB/s)	TED(s)
1	315.9886	6329.9436	1003.9496	227.1512	28	50	155.00	557.75	357.52
2	335.3631	6113.9670	979.2256	227.0584	28	50	152.50	577.55	349.80
3	354.5489	7048.3813	919.2507	114.1853	28	67	152.70	429.31	366.43
4	36.0993	3239.7275	488.2076	130.6181	25	28	225 .98	467.63	282.51
5	Não foi possível gerar pois excede a quantidade de memória local disponível no FPGA								

Nota – K - *kernel*, T sw - tempo do *kernel* em software, T hw - tempo do *kernel* em hardware, T F-C - tempo de cópia Fortran para C, T C-F - tempo de cópia do C para o Fortran, EL - elementos lógicos, R - RAM blocks, F - frequência máxima, B - bandwidth, TED = tempo de execução da dinâmica

Figura 38 – Execução dinâmica dos *kernels* do módulo advection para o modelo Task

6.3.6 Kernels *advect* implementados em NDRange

A Tabela 13 apresenta os dados obtidos da execução dos *kernels* do grupo *advect* implementados em NDRange. Nessa implementação cada *kernel* possui 30 *work-items* nas dimensões x e y e 4 unidades de computação. Ao analisar a tabela observa-se que diferente das implementações em NDRange dos outros grupos que possuem um melhor desempenho no tempo de execução da dinâmica, se comparado com sua versão em Task a implementação NDRange do grupo *advect* produziu um resultado contrário para os *kernels* 1, 2 e 3 e 5. Nessa situação a implementação desses *kernels* fazendo cópia dos dados (menos o 5) para a memória local no modelo de programação Task melhora o tempo de execução da dinâmica. A exceção no grupo foi o *kernel* 4, cuja implementação em NDRange obteve um melhor desempenho se comparada sua implementação em Task que pode ser explicado pelo aumento da largura de banda ao acessar a memória e a divisão do processamento nas unidades de computação geradas.

Tabela 13 – *Kernels* *advect* para o modelo de programação NDRange para 4 CU e WI(30,30) na resolução 35X60X60

K	T sw(us)	T hw(us)	T F-C(us)	T C-F(us)	EL(%)	R(%)	F(MHz)	B(MB/s)	TED(s)
1	315.9886	12349.2895	707.3886	192.7074	41	50	202.79	289.99	464.44
2	335.3631	11339.6767	864.6310	209.8474	41	50	205.50	313.20	444.30
3	354.5489	10549.9137	445.0725	161.8571	41	53	209.55	306.00	408.99
4	36.0993	2989.5611	308.3724	58.1271	29	23	215.62	506.11	193.36
5	586.0613	27538.9855	733.6886	765.5406	38	34	200.00	244.68	166.14

Nota – CU - *Compute Unit*, K - *kernels*, T sw - tempo do *kernel* em software, T hw - tempo do *kernel* em hardware, T F-C - tempo de cópia *Fortran* para C, T C-F - tempo de cópia do C para o *Fortran*, EL - elementos lógicos, R - RAM *blocks*, F - frequência máxima, B - *bandwidth*, TED = tempo de execução da dinâmica

A Tabela 14 refere-se a implementação dos *kernels* do grupo *advect* para execução da dinâmica na resolução 35X100X100. Os *kernels* foram codificados para gerar 50 *work-items* nas dimensões x e y e 2 unidades de computação. Nesse estudo de caso ainda buscou-se comprovar alguma relação na qual aumentando a quantidade de dados a serem processados o hardware melhoraria o desempenho, mais uma vez isso não aconteceu, considerando que o tempo de execução da dinâmica na resolução 35X100X100 usando apenas a CPU é de 320.5s. Nenhum dos *kernels* conseguiu igualar ou superar esse tempo.

Para o módulo *advect* seguindo o modelo NDRange a Figura 39 ilustra uma visão geral dos tempos obtidos para a execução dinâmica dos *kernels*.

6.3.7 Kernels *acustic* implementados em Task

Esse foi o último grupo de *kernels* a ser implementado. Seguindo o padrão dos estudos de caso adotados nesta tese, foram fornecidas duas implementações em Task para os *kernels* desse grupo, considerando a cópia de dados para a memória local ou não.

Tabela 14 – *Kernels* advect para o modelo de programação NDRange para 2 CU e WI(50,50) na resolução 35X100X100

K	T sw(us)	T hw(us)	T F-C(us)	T C-F(us)	EL(%)	R(%)	F(MHz)	B(MB/s)	TED(s)
1	821.6232	23790.2495	1760.6772	530.1061	32	32	210.00	411.99	1103.76
2	884.1366	22417.3131	1727.6318	544.8861	32	32	220.00	437.21	1057.96
3	958.4181	18597.6627	1679.5527	300.0974	32	34	210.00	451.73	957.81
4	215.1467	9036.9023	622.1911	159.6545	26	19	228.57	464.82	568.02
5	1782.3376	34326.2409	1383.7886	2259.9634	38	34	202.00	546.32	405.65

Nota – CU - *Compute Unit*, K - *kernels*, T sw - tempo do *kernel* em software, T hw - tempo do *kernel* em hardware, T F-C - tempo de cópia *Fortran* para C, T C-F - tempo de cópia do C para o *Fortran*, EL - elementos lógicos, R - RAM *blocks*, F - frequência máxima, B - *bandwidth*, TED = tempo de execução da dinâmica

Figura 39 – Execução dinâmica dos *kernels* do módulo advect para o modelo NDRange

Fonte: Elaborada pelo autor.

A Tabela 15 apresenta os dados obtidos para execução dos *kernels* que não fazem cópia dos dados para a memória local, confirmando que o modelo Task de programação que lê e processa os dados da memória global não produz um bom desempenho analisando os dados produzidos no escopo deste trabalho. Por sua vez, quando os *kernels* são implementados em Task e é possível copiar os dados a serem processados para a memória local do FPGA, o hardware gerado produz o melhor desempenho considerando a execução da dinâmica do BRAMS que foi o objeto de estudo neste trabalho.

Os dados apresentados na Tabela 16 são referentes aos *kernels* do grupo acoustic que puderam ser gerados fazendo cópia dos dados a serem processados para a memória local. É

Tabela 15 – *Kernels* acoustic para o modelo de programação Task sem cópia para memória local na resolução 35X60X60

K	T sw(us)	T hw(us)	T F-C(us)	T C-F(us)	EL(%)	R(%)	F(MHz)	B(MB/s)	TED(s)
1	888.2771	222667.5021	421.4863	498.8550	37	34	210.70	183.96	442.37
2	549.5886	35487.9103	1035.6594	61.7591	38	82	168.63	283.97	261.51
3	72.7809	12725.5491	572.0131	123.5344	27	21	215.62	633.82	209.55
4	452.5411	151488.5544	666.0910	190.6529	30	30	218.00	161.07	778.28
5	105.3337	140360.6732	445.7917	112.3953	25	19	241.66	11.00	732.70

Nota – K - *kernel*, T sw - tempo do *kernel* em software, T hw - tempo do *kernel* em hardware, T F-C - tempo de cópia *Fortran* para C, T C-F - tempo de cópia do C para o *Fortran*, EL - elementos lógicos, R - RAM *blocks*, F - frequência máxima, B - *bandwidth*, TED = tempo de execução da dinâmica

possível verificar com os dados que os *kernels* 3, 4 e 5 reduziram o tempo de processamento da dinâmica aproximadamente pela metade. Os *kernels* 1 e 2 não puderam ser gerados, pois não havia memória local (on-chip) suficiente para copiar os dados.

Tabela 16 – *Kernels* acoustic para o modelo de programação Task sem cópia para memória local na resolução 35X60X60

K	T sw(us)	T hw(us)	T F-C(us)	T C-F(us)	EL(%)	R(%)	F(MHz)	B(MB/s)	TED(s)
1	Não foi possível gerar pois excede a quantidade de memória local disponível no FPGA								
2	Não foi possível gerar pois excede a quantidade de memória local disponível no FPGA								
3	72.7809	3739.1259	578.0896	119.7335	30	59	211.00	2163.66	176.11
4	452.5411	25902.4713	694.0946	210.7913	33	82	147.91	942.90	261.98
5	105.3337	6920.0830	405.0723	118.7783	27	48	237.50	218.8870	185.59

Nota – K - *kernel*, T sw - tempo do *kernel* em software, T hw - tempo do *kernel* em hardware, T F-C - tempo de cópia *Fortran* para C, T C-F - tempo de cópia do C para o *Fortran*, EL - elementos lógicos, R - RAM *blocks*, F - frequência máxima, B - *bandwidth*, TED = tempo de execução da dinâmica

Para o módulo acoustic seguindo o modelo Task a Figura 40 ilustra uma visão geral dos tempos obtidos para a execução dinâmica dos *kernels*.

6.3.8 Kernels acoustic implementados em NDRange

Duas versões foram implementadas para os *kernels* do grupo acoustic em NDRange. Uma para executar a dinâmica na resolução 35X60X60 e outra para execução na resolução 35X100X100. Exceto o *kernel* 2 deste grupo, comparando este grupo com os demais, todos os outros *kernels* não tiveram uma distância muito grande comparando o tempo total de execução da dinâmica com o tempo da dinâmica sendo executada apenas pela CPU. Analisando a Tabela 17 um fator evidente é a largura de banda (B) alcançada pelos *kernels* desse grupo, que foi melhor do que os demais. Esse padrão se repete para a versão produzida para executar a dinâmica na resolução 35X100X100, que é apresentada na Tabela 18. Destaca-se que esse grupo de *kernels*, de modo geral, é chamado 6 vezes menos que os *kernels* dos grupos advect, grad e turbulência.

Figura 40 – Execução dinâmica dos *kernels* do módulo acoustic para o modelo Task Tabela 17 – *Kernels* acoustic para o modelo de programação NDRange para 4 CU e WI(30,30) na resolução 35X60X60

K	T sw(us)	T hw(us)	T F-C(us)	T C-F(us)	EL(%)	R(%)	F(MHz)	B(MB/s)	TED(s)
1	888.2771	13986.9341	417.1947	503.3183	48	46	196.29	2929.80	146.66
2	549.5886	23497.4938	1007.9135	111.4076	40	47	198.33	429.16	254.12
3	72.7809	5953.7012	479.6610	106.3234	35	29	211.90	1357.59	183.77
4	452.5411	5395.0968	596.1932	114.6581	45	53	200.00	4528.53	131.77
5	105.3337	2132.3476	406.1741	68.4372	29	26	260.00	709.09	116.94

Nota – CU - *Compute Unit*, K - *kernels*, T sw - tempo do *kernel* em software, T hw - tempo do *kernel* em hardware, T F-C - tempo de cópia *Fortran* para C, T C-F - tempo de cópia do C para o *Fortran*, EL - elementos lógicos, R - RAM *blocks*, F - frequência máxima, B - *bandwidth*, TED = tempo de execução da dinâmica

Para o módulo Acoustic seguindo o modelo NDRange a Figura 41 ilustra uma visão geral dos tempos obtidos para a execução dinâmica dos *kernels*.

O gráfico ilustrado na Figura 42 apresenta de forma resumida os *kernels* implementados em NDRange que obtiveram melhores tempos para a execução do módulo da dinâmica.

Tabela 18 – *Kernels* acustic para o modelo de programação NDRange para 4 CU e WI(50,50) e resolução 35X100X100

K	T sw(us)	T hw(us)	T F-C(us)	T C-F(us)	EL(%)	R(%)	F(MHz)	B(MB/s)	TED(s)
1	2502.7639	38340.7493	850.1487	850.1487	48	46	202.00	2971.53	402.38
2	1535.9986	60363.0193	1994.3364	284.3417	40	47	198.33	463.93	620.80
3	222.5388	12259.7135	932.6324	279.5367	35	29	210.00	1827.57	420.09
4	1286.6388	16068.6641	1152.3959	339.2709	45	53	198.33	4221.97	390.75
5	367.3318	7673.0973	742.3593	192.3975	29	32	236.00	547.37	353.29

Nota – CU - *Compute Unit*, K - *kernels*, T sw - tempo do *kernel* em software, T hw - tempo do *kernel* em hardware, T F-C - tempo de cópia *Fortran* para C, T C-F - tempo de cópia do C para o *Fortran*, EL - elementos lógicos, R - RAM *blocks*, F - frequência máxima, B - *bandwidth*, TED = tempo de execução da dinâmica

Figura 41 – Execução dinâmica dos *kernels* do módulo acustic para o modelo NDRange

Fonte: Elaborada pelo autor.

6.4 Estudo de caso II

Para este estudo de caso apresentamos em relação a medida tempo de execução apenas o tempo total de execução da dinâmica em conjunto com o *kernel* executado. Nesta fase foram implementados 19 *kernels*. Para cada um dos *kernels* foram fornecidas implementações considerando os modelos de programação (*NDRange*) do Intel *OpenCL* e acesso aos dados na memória global. Com base nos resultados do primeiro estudo de caso o acesso aos dados da memória global foi otimizado de modo a permitir em alguns casos uma maior taxa na *bandwidth* e no padrão de acesso aos dados.

As melhorias implementadas foram realizadas ao observar que o acesso aos dados na memória ocorriam de forma irregular devido a transformação direta do código *Fortran* para *OpenCL*. Como *Fortran* inicia os índices das matrizes em 1 e *OpenCL* em 0 na conversão direta

Figura 42 – Execução dos *kernels* com melhores tempos do módulo da dinâmica no estudo de caso I

Fonte: Elaborada pelo autor.

isso ocasionava um acesso não otimizado aos dados na memória devido ao hardware gerado pelo compilador *OpenCL*.

Além da alteração no acesso os dados na memória, também foi utilizado o desenrolamento parcial dos loops (*loop unrolling*) de computação dos *kernels*. Ao desenrolar parcialmente os loops foi possível aglutinar os acessos a memória, uma vez que eles são consecutivos sobre o índice k presente no *loop* de computação do *kernel*, resultando em um uso eficiente da taxa de *bandwidth* da memória.

Além disso, no estudo de caso I observou-se que ao replicar o hardware em NDRange, ou seja, aumentar o número de unidades de computação, a taxa de *bandwidth* reduzia, pois as unidades de computação competem pelo acesso à memória. Essa competição gerava o *overhead* no controlador da memória e dessa forma, o tempo de disponibilização dos dados era grande para cada unidade de computação. Nessa nova implementação foi informado ao compilador apenas a quantidade de *work-items* 20X20. Não foi especificado valor para o número de unidades de computação. Nesse caso, por padrão o compilador assume que será gerada apenas 1 unidade de computação.

Os *kernels* implementados foram divididos em 4 grupos, considerando o módulo da dinâmica do BRAMS, são eles: *turbulência*, *grad*, *advect* e *acustic*. Além das características do hardware, o código da dinâmica foi executado para a resolução de grade 35X60X60. Nas próximas subseções são apresentados os resultados para cada um dos grupos.

6.4.1 Kernels da turbulência implementados em NDRange

Como ilustrado na Tabela 19, a taxa de *bandwidth* melhorou significativamente devido ao novo padrão de acesso aos dados da memória. Isso também permitiu que o tempo de execução

dos *kernels* fosse reduzido.

A implementação do *kernel* 1 no estudo de caso I apresentou um tempo de 251.9 segundos como o melhor resultado, isso significava mais que o dobro do tempo para execução do código da dinâmica utilizando apenas a CPU. Nessa nova implementação o tempo foi de 165.3 segundos. Uma redução de quase 100 segundos em relação ao melhor tempo para este *kernel* até a execução do estudo de caso II. O mesmo pode ser observado para os *kernels* 2, 3 e 4, que também apresentaram melhoria significativa nos tempos ao modificar o padrão de acesso aos dados e aplicar o desenrolamento de loops.

O desenrolamento parcial dos *loops* não foi aplicado nas implementações anteriores, pois excedia a quantidade de elementos lógicos disponíveis na placa Arria 10. Nessa nova implementação foi possível aplicar, pois o hardware não foi replicado em virtude da competição gerada durante o acesso à memória. O fator de desenrolamento aplicado para todos os *kernels* desse grupo foi de 8.

Tabela 19 – *Kernels* turbulência na resolução 35X60X60

K	Tempo (s)	Power (W)	Energia (J)	F (MHz)	EL	R (blocks)	DSP	B (MB/s)	Gflops
1	165.3	22.50	3719.25	184.46	54%	63%	7%	968.07	0.4945
2	143.4	20.78	2979.85	219.34	40%	27%	15%	3198.37	0.7720
3	110.4	21.47	2370.29	225.58	38%	35%	19%	6421.85	0.1192
4	114.3	20.80	2377.44	211.90	35%	28%	8%	2371.43	0.8178

Nota – K - *kernel*, Tempo - tempo CPU+FPGA, F - frequência máxima, EL - elementos lógicos, R - RAM blocks, B - bandwidth

Também foi possível medir o consumo real de energia para os *kernels* desenvolvidos neste estudo de caso. A medição tornou-se viável com a disponibilização de uma aplicação escrita na linguagem C por parte do HARP. O código fonte dessa aplicação também foi anexado ao *framework* de modo que é possível habilitar a medição do consumo de energia durante a execução da dinâmica do BRAMS, fazendo uso do *kernel*. Os dados do consumo de energia também são apresentados na Tabela 20 para os *kernels* da turbulência.

Faz-se importante destacar que a plataforma HARP não disponibiliza uma forma de medir o consumo de energia apenas do processador, de modo que o consumo foi estimado usando o Thermal Design Power (TDP) do processador. Na Tabela 20 são apresentados os valores para o consumo de energia, considerando a medição real durante a execução do *kernel* e o valor do TDP da placa e do processador Xeon. Em ambas as medições o FPGA é mais eficiente do ponto de vista do consumo de energia.

O gráfico ilustrado na Figura 43 apresenta de forma resumida dados sobre a eficiência de energia dos *kernels* da turbulência em relação ao processador Xeon.

Tabela 20 – Eficiência de energia dos *kernels* da turbulência na Arria 10 em relação a execução na CPU

	Tempo (s)	Power (W)	TDP (W)	Energia (J)	Energia (J) TDP
Turb 1	165.3	22.5	70	3719.25	11571.00
Turb 2	143.4	20.78	70	2979.85	10038.00
Turb 3	110.4	21.47	70	2370.29	7728.00
Turb 4	114.3	20.8	70	2377.44	8001.00
Xeon - E5-2600 v4	108.6	-	160	-	17376.00

Figura 43 – Consumo de energia (J) considerando o TDP

Fonte: Elaborada pelo autor.

6.4.2 Kernels *grad* implementados em *NDRange*

Esse grupo também apresentou uma melhora significativa no tempo de execução dos *kernels* em conjunto com a dinâmica. O caso a ser observado é o *kernel* 4, cujo melhor tempo de execução em conjunto com a dinâmica foi 193,01 segundos. Com a implementação no estudo de caso II o seu tempo sofreu piora. Isso pode ser explicado, em partes, pelo fator de desenrolamento aplicado a este *kernel*, que foi de 4, enquanto os outros *kernels* desse mesmo grupo foi aplicado fator 8. Outra causa que contribuiu para essa degradação é que ao alterar a iteração dos *loops*, a melhora de desempenho não foi observada como nos outros *kernels*, merecendo uma melhor análise do hardware gerado. Para todos os *kernels* desse grupo foi especificado para o compilador apenas o número de *work-items* desejado 20X20. Na Tabela 21 são apresentados os dados obtidos para esse grupo de *kernels* nessa fase.

Com os dados de *power* obtidos durante a execução dos *kernels* foi possível medir a eficiência de energia para os *kernels* desse grupo em relação ao processador Xeon. Os dados dessa medição são apresentados na Tabela 22 e consideram o valor real do consumo de potência

Tabela 21 – *Kernels* grad – Resolução 35x60x60

K	Tempo (s)	Power (W)	Energia (J)	F (MHz)	EL	R (blocks)	DSP	B (MB/s)	Gflops
1	165.5	21.37	3536.7	149.72	51%	52%	15%	634.74	1.1491
2	165.9	25.63	4252.02	214.63	52%	53%	16%	620.68	1.0054
3	153.8	22.40	3445.12	192.82	51%	54%	16%	819.79	1.7185
4	216.6	21.79	4719.71	198.49	46%	50%	8%	315.53	0.9021
5	110.8	19.97	2212.68	240.61	27%	18%	4%	6813.18	1.0573

Nota – k - *kernel*, Tempo - tempo CPU+FPGA, F - frequência máxima, EL - elementos lógicos, R - RAM blocks, B - *bandwidth*

(*power*) medido para o FPGA e o valor do TDP para o FPGA e o processador Xeon. Para esse grupo também é possível afirmar que em ambas as medições o FPGA é mais eficiente do ponto de vista do consumo de energia.

O gráfico ilustrado na Figura 44 apresenta de forma resumida a relação de eficiência de energia dos *kernels* desse grupo em relação ao processador Xeon.

Tabela 22 – Eficiência de energia dos *kernels* grad na Arria 10 em relação a execução na CPU

	Tempo (s)	Power (W)	TDP (W)	Energia (J)	Energia (J) TDP
Grad 1	165.5	21.37	70	3536.74	11585.00
Grad 2	165.9	25.63	70	4252.02	11613.00
Grad 3	153.8	22.40	70	3445.12	10766.00
Grad 4	216.6	21.79	70	4719.71	15162.00
Grad 5	110.8	19.97	70	2212.68	7756.00
Xeon - E5-2600 v4	108.6	-	160	-	17376.00

6.4.3 *Kernels advect implementados em NDRange*

Esse grupo apresentou uma melhora de quase 50% para todos os *kernels* no tempo de execução em conjunto com a dinâmica. Para todos os *kernels* desse grupo foi especificado para o compilador apenas o número de *work-items* desejado, 20X20. Com exceção do *kernel* 5 implementado com o fator 4 (valor maior ultrapassa o limite de tamanho do FPGA) para desenrolamento parcial dos loops de computação, os demais *kernels* desse grupo foram implementados com o fator 8. Na Tabela 23 são apresentados os dados obtidos para esse grupo de *kernels* nesta fase.

Na Tabela 24 são apresentados os dados de energia para esse grupo de *kernels*. Os valores consideram o real consumo de potência (*power*) medido para o FPGA e o valor do TDP para o FPGA e o processador Xeon. Para esse grupo também é possível afirmar que em ambas as medições o FPGA é mais eficiente do ponto de vista do consumo de energia.

Figura 44 – Consumo de energia (J) considerando o TDP

Fonte: Elaborada pelo autor.

Tabela 23 – Kernels advect na resolução 35x60x60

K	Tempo (s)	Power (W)	Energia (J)	F (MHz)	EL	R (blocks)	DSP	B (MB/s)	Gflops
1	226.6	23.35	5291.11	196.46	40%	49%	4%	920.00	0.5020
2	221.1	23.41	5175.95	193.42	39%	49%	5%	929.75	0.5106
3	214.6	23.09	4955.11	211.19	37%	41%	4%	757.00	0.8622
4	128.3	19.29	2474.91	229.35	27%	18%	2%	2802.14	0.5797
5	116.3	22.63	2631.87	197.78	58%	41%	21%	1797.77	1.1149

Nota – K - kernel, Tempo - tempo CPU+FPGA, F - frequência máxima, EL - elementos lógicos, R - RAM blocks, B - bandwidth

Tabela 24 – Eficiência de energia dos *kernels* grad na Arria 10 em relação a execução na CPU

	Tempo(s)	Power(W)	TDP	Energia(J)	Energia(J) TDP
Advect 1	226.6	23.35	70	5291.11	15862.00
Advect 2	221.1	23.41	70	5175.95	15477.00
Advect 3	214.6	23.09	70	4955.11	15022.00
Advect 4	128.3	19.29	70	2474.91	8981.00
Advect 5	116.3	22.63	70	2631.87	8141.00
Xeon - E5-2600 v4	108.6	-	160	-	17376.00

O gráfico ilustrado na Figura 45 apresenta de forma resumida a relação de eficiência de energia dos *kernels* desse grupo em relação ao processador Xeon. É possível verificar que os *kernels* 1, 2 e 3 possuem um consumo próximo ao apresentado pela CPU, mas isso ocorre se for considerado o cálculo de energia utilizando o TDP. De acordo com a medição real de *power* para esses *kernels*, a diferença de consumo em relação a CPU é maior.

Figura 45 – Consumo de energia (J) considerando o TDP

Fonte: Elaborada pelo autor.

6.4.4 Kernels acustic implementados em NDRange

Em relação a melhora obtida pelos outros grupos, esse grupo apresentou uma melhora mais modesta do ponto de vista do tempo de execução da dinâmica em conjunto com os *kernels*. Houve uma redução média de 30 segundos considerando o tempo de execução de todos os *kernels* desse grupo em relação ao melhor tempo obtido por cada um deles no estudo de caso I. Mesmo sendo uma melhora modesta, esse resultado também reflete no consumo de energia. Este é um tempo aceitável para uma simulação de previsão do tempo.

Para todos os *kernels* desse grupo foi especificado para o compilador apenas o número de *work-items* desejado, 20X20. A melhora modesta obtida nesse grupo é justificada pelo fator de desenrolamento dos *loops* de computação. Para o *kernel* 1 foi aplicado o fator 2. Para o *kernel* 2 não foi possível aplicar nenhum fator, pois quando um fator era aplicado não era possível gerar o hardware para o mesmo, pois este excedia o tamanho da lógica disponível no FPGA. Para o *kernel* 3 foi aplicado o fator 8. Essa melhora se refletiu na redução do tempo de execução em relação ao tempo obtido no estudo de caso I. Para o *kernel* 4 foi aplicado um fator de 4 também devido ao tamanho do *kernel* no momento da síntese. Por fim, o *kernel* 5 teve um fator de 8 aplicado, no entanto, esse foi o único *kernel* desse grupo com a necessidade de usar uma barreira, devido a alteração no algoritmo de modo a garantir que o resultado final fosse produzido corretamente. O uso de barreiras em *OpenCL* não é recomendado, pois degrada o desempenho do hardware gerado.

Na Tabela 25 são apresentados os dados obtidos para esse grupo de *kernels*.

Tabela 25 – *Kernels* acoustic

<i>kernels</i> acoustic na resolução 35x60x60									
K	Tempo (s)	Power (W)	Energia (J)	F (MHz)	EL	R (blocks)	DSP	B (MB/s)	Gflops
1	120.9	22.38	2705.74	177.27	52%	54%	7%	4948.16	0.2507
2	195.6	21.42	4189.75	187.33	38%	49%	6%	512.30	0.5471
3	123.3	19.89	2452.44	277.77	27%	18%	<1%	2736.46	0.2556
4	126.2	20.39	2573.22	226.19	31%	25%	2%	6425.22	0.4389
5	118.8	21.75	2583.90	225.66	32%	57%	1%	731.98	0.1009

Nota – K - *kernel*, Tempo - tempo CPU+FPGA, F - frequência máxima, EL - elementos lógicos, R - RAM blocks, B - *bandwidth*

A Tabela 26 apresenta os dados de consumo de energia para esse grupo de *kernels* em relação a CPU. Os valores de energia consideram o valor real do consumo de potência (*power*) para os *kernels* executando no FPGA e o valor do TDP para o FPGA e o processador Xeon. Para esse grupo também é possível afirmar que em ambas as medições o FPGA é mais eficiente do ponto de vista do consumo de energia.

Tabela 26 – Eficiência de energia dos *kernels* acoustic na Arria 10 em relação a execução na CPU

	Tempo(s)	Power(W)	TDP	Energia(J)	Energia(J) TDP
Acoustic 1	120.9	21.37	70	2583.63	8463.00
Acoustic 2	195.6	25.63	70	5013.23	13692.00
Acoustic 3	123.3	22.40	70	2761.92	8631.00
Acoustic 4	126.2	21.79	70	2749.90	8834.00
Acoustic 5	118.8	19.97	70	2372.44	8316.00
Xeon - E5-2600 v4	108.6	-	160	-	17376.00

O gráfico ilustrado na Figura 46 apresenta de forma resumida a relação de eficiência de energia dos *kernels* desse grupo em relação ao processador Xeon.

O gráfico de barras apresentado na Figura 47 mostra os tempos de execução CPU-FPGA da dinâmica.

6.5 Estudo de caso III

Neste estudo de caso foram compilados os 19 *kernels* implementados, no entanto, para a Stratix V foi possível gerar 17 devido a erros de compilação no processo de geração do hardware. Os *kernels* foram executados em conjunto com a dinâmica do BRAMS na plataforma disponível no LCR, constituída por um processador Intel Xeon-E5-1603 v3 2.8 GHz, Sistema Operacional CentOS versão 7 e placa BittWare S5PH-Q, que dispõe do FPGA Stratix V. Os tempos de execução apresentados na sequência foram obtidos com base em uma média de 30 execuções para cada *kernel*.

Figura 46 – Consumo de energia (J) considerando o TDP

Figura 47 – Execução dos *kernels* com melhores tempos do módulo da dinâmica do estudo de caso II

O tempo total de execução da dinâmica apenas na CPU desta plataforma foi de 182.5 segundos. Esse tempo é superior ao da execução da dinâmica na plataforma HARP, também executando apenas em CPU. De modo análogo ao que aconteceu na plataforma HARP, nenhum dos *kernels* conseguiu reduzir o tempo de execução da dinâmica usando CPU e FPGA. A eficiência de energia também foi melhor para esta plataforma.

6.5.1 Kernels turbulência Stratix V

No grupo da turbulência, de acordo com a Tabela 27, a frequência máxima para os *kernels* executados na Stratix V foi maior, exceto para o *kernel* 4, em relação ao mesmo grupo de *kernels* executados na Arria 10. Porém, na Stratix somente o *kernel* 4 consumiu mais (36% na Stratix e 35% na Arria 10) elementos lógicos que os demais *kernels* executados na Arria 10. Além disso, apenas o *kernel* 3, dentre os *kernels* do grupo da turbulência, obteve maior *bandwidth* na Stratix V do que na Arria 10. A maioria dos *kernels* na Stratix V utilizou mais DSPs do que na Arria 10.

Tabela 27 – Kernels turbulência executados na Stratix V na resolução 35x60x60

K	Tempo (s)	Power (W)	Energia (J)	F (MHz)	EL	R (blocks)	DSP	B MB/s	Gflops
1	277.2	15.54	4307.688	287.02	35%	32%	11%	1197.40	0.6103
2	267.2	12.85	3433.52	317.96	28%	18%	13%	3443.93	0.8313
3	193.4	12.52	2421.368	327.76	23%	18%	23%	4193.50	0.6941
4	200.4	16.04	3214.416	293.94	36%	27%	37%	2487.54	0.8578

Nota – K - *kernel*, Tempo - tempo CPU+FPGA, F - frequência máxima, EL - elementos lógicos, R - RAM blocks, B - *bandwidth*

6.5.2 Kernels grad Stratix V

A Tabela 28 mostra os dados para o grupo grad. Para esse grupo de *kernels*, a frequência máxima obtida foi maior para todos os *kernels* executados na Stratix V em relação a Arria 10. A exceção foi o *kernel* 2, que apresentou frequência 213,17 e na Arria obteve 214,63. Os *kernels* compilados para a Stratix V consumiram mais elementos lógicos do que na Arria 10. Apenas o *kernel* 5 consumiu menos no comparativo, 22% na Stratix V e 27% na Arria 10. Para esse grupo a Stratix V também utilizou mais DSPs do que a Arria 10.

Tabela 28 – Kernels grad executados na Stratix V na resolução 35x60x60

K	Tempo (s)	Power (W)	Energia (J)	F (MHz)	EL	R (blocks)	DSP	B MB/s	Gflops
1	262.1	21.82	5719.02	236.07	54%	44%	35%	757.22	1.3709
2	272.8	21.98	5996.14	213.17	55%	44%	36%	468.68	0.7590
3	263.3	21.97	5784.70	206.44	55%	46%	36%	742.32	1.5557
4	303.3	23.50	7127.55	231.16	57%	55%	38%	390.09	1.1148
5	189.4	12.19	2308.79	294.29	22%	17%	20%	2840.46	0.8690

Nota – K - *kernel*, Tempo - tempo CPU+FPGA, F - frequência máxima, EL - elementos lógicos, R - RAM blocks, B - *bandwidth*

6.5.3 Kernels *advect Stratix V*

A Tabela 29 mostra os dados para o grupo *advect*. Para esse grupo de *kernels* a frequência máxima obtida foi maior para todos os *kernels* executados na Stratix V em relação a Arria 10. Observa-se nesse grupo um certo equilíbrio quanto ao número de elementos lógicos utilizados na compilação dos *kernels* para os FPGAs Stratix V e Arria 10. Esse grupo na Stratix V também utilizou mais DSPs do que a Arria 10.

Tabela 29 – Kernels *advect* executados na Stratix V na resolução 35x60x60

K	Tempo (s)	Power (W)	Energia (J)	F (MHz)	EL	R (blocks)	DSP	B MB/s	Gflops
1	339.4	16.50	5600.10	275.02	40%	37%	19%	957.33	0.5257
2	331.9	16.55	5492.95	261.50	41%	37%	20%	1013.08	0.5564
3	236.4	16.88	3990.43	257.86	37%	37%	17%	2698.54	0.6274
4	231.7	12.01	2782.72	306.56	24%	16%	7%	2905.47	0.6011

Nota – K - *kernel*, Tempo - tempo CPU+FPGA, F - frequência máxima, EL - elementos lógicos, R - RAM blocks, B - *bandwidth*

6.5.4 Kernels *acustic Stratix V*

A Tabela 30 apresenta os dados para o grupo *acustic*. Para esse grupo de *kernels* a frequência máxima obtida foi maior para todos os *kernels* executados na Stratix V em relação a Arria 10. De modo geral, isso ocorreu para todos os grupos de *kernels* compilados para a Stratix V. Houve um equilíbrio quanto ao número de elementos lógicos utilizados na compilação dos *kernels* para os FPGAs Stratix V e Arria 10. Esse grupo também utilizou mais DSPs na Stratix V do que na Arria 10, a exceção foi o *kernel* 2 que utilizou mais DSPs na Arria 10.

Tabela 30 – Kernels *acustic* executados na Stratix V na resolução 35x60x60

K	Tempo (s)	Power (W)	Energia (J)	F (MHz)	EL	R (blocks)	DSP	B MB/s	Gflops
1	193.9	18.55	3596.85	199.00	61%	80%	34%	7330.81	0.3715
3	205.3	11.17	2293.20	327,54	19%	16%	3%	665.95	0.2489
4	208.3	12.71	2647.49	303,20	24%	21%	8%	872.12	0.4766
5	196,2	15.33	3007.75	288.85	28%	40%	5%	704.88	0.0972

Nota – K - *kernel*, Tempo - tempo CPU+FPGA, F - frequência máxima, EL - elementos lógicos, R - RAM blocks, B - *bandwidth*

Por fim, a Figura 48 ilustra um gráfico de eficiência de energia da Stratix V em relação a Arria 10 executando os mesmos *kernels*. Os dados de *power* para a Stratix V foram estimados usando a ferramenta *quartus_{pow}*, disponível no ambiente de compilação do Intel *OpenCL* SDK.

Figura 48 – Eficiência de energia para os *kernels* executados na Stratix V e na Arria 10

Fonte: Elaborada pelo autor.

6.6 Discussão dos resultados

Neste capítulo foram apresentados e discutidos os resultados dos estudos de caso conduzidos a partir da aplicação do *framework* para coprojeto hardware/software proposto ao módulo da dinâmica do BRAMS.

Os resultados apresentados resultam dos dois modelos de programação implementados Task e NDRange. Eles sugerem que o modelo de programação que mais se aproxima em alcançar um ganho de desempenho em relação a CPU é o NDRange para o caso da dinâmica do BRAMS. Infelizmente, devido a restrições de equipamento para compilação dos *kernels* implementados não foi possível explorar outras formas de otimização do código *OpenCL* gerado. Em alguns casos, o algoritmo original do *kernel* implementado em *Fortran 90*, que foi convertido em *OpenCL* não permitiu explorar um maior grau de paralelismo, uma característica que é muito bem explorada pelo FPGA.

Outro fator a ser considerado é que devido ao tamanho do FPGA, não foi possível fornecer uma implementação em NDRange que fizesse cópia dos dados para a memória local. Tentativas nesse sentido incluíram por exemplo, copiar parte dos dados, executar e escrever na memória global. Elas degradavam ainda mais a execução do sistema, sugerindo uma solução ruim.

O desempenho obtido pelo desenrolamento depende em grande parte de quão grande é a quantidade de iterações do *loop*. Se for muito pequeno, a melhoria de desempenho será quase imperceptível. Além disso, o valor do fator aplicado deve ser uma potência de dois, pois o tamanho das portas de acesso à memória geradas pelo compilador é sempre uma potência de dois. Ao usar fatores de desenrolamento que não sejam potências de dois, o compilador ainda gerará a porta de memória com o tamanho mais próximo de uma potência de dois. Isso mascara as transações de memória extra, resultando em um grande desperdício de *bandwidth* da memória

externa.

Mesmo com as melhorias aplicadas resultando em melhor desempenho, de modo geral, para todos os *kernels* desenvolvidos o código gerado para ser executado no FPGA (geração atual) ainda é mais lento se comparado com um código altamente otimizado em uma CPU ou GPU moderna. Isso também ocorre devido a largura de banda da memória externa do FPGA, que é muito menor.

É importante destacar que a versão do compilador Intel *OpenCL* usado para geração do hardware para as duas placas foram diferentes. Para a Stratix V foi utilizado o 16.1 e para a Arria 10 o 16.0. O mesmo compilador não pôde ser utilizado para a geração do hardware de ambas as placas pois a versão do BSP fornecido era para as versões dos respectivos compiladores. Desse modo, pode-se observar que para a Stratix V os recursos lógicos disponíveis no FPGA foram melhor utilizados para a maioria dos *kernels* em relação a Arria 10.

Vale ressaltar que o dispositivo Arria 10 dispõe de aproximadamente o dobro de elementos lógicos (ELs) e quase seis vezes mais DSPs em comparação com o FPGA Stratix V. Contudo, de acordo com os testes realizados o FPGA Stratix V apresentou um desempenho proporcionalmente equiparado ao Arria 10. Uma desvantagem para a Stratix V nos testes realizados foi a versão do *OpenCL* para o código do host. Para a Arria 10 foi utilizado a 2.0 e para a Stratix V (a placa utilizada não suporta a versão 2.0) a versão 1.0. Considerando o processo de transferência de dados para a memória global a versão 2.0 é mais otimizada, pois o processamento dos dados pode ser iniciado enquanto a cópia dos dados está em andamento. Na versão 1.0 é necessário que o processo de cópia seja finalizado para que a computação pelo *kernel* seja iniciada.

Apesar dos resultados dos *kernels* não terem igualado nem superado o tempo de execução da dinâmica usando apenas CPU, o FPGA pode e deve ser aplicado ao código do BRAMS, pois os tempos obtidos são factíveis para executar uma simulação de previsão do tempo. Além disso, o BRAMS executado em um cluster no CPTEC/INPE exige um alto consumo de energia e uma solução em FPGA é altamente atraente. Como foi demonstrado neste trabalho, o consumo de energia por parte dos *kernels* executando no FPGA é muito baixo se comparado com uma CPU moderna, como por exemplo, o Xeon.

Todos os códigos em *Fortran* que foram convertidos em *OpenCL* nesse trabalho encontram-se em produção e execução para realização da previsão do tempo do Brasil. Trata-se de uma aplicação real e não de *benchmarks* que são feitos e ou adaptados para explorar o melhor desempenho do dispositivo alvo.

Nesse sentido, explorar formas de otimizar as implementações produzidas nesta tese deve se concentrar em obter um melhor desempenho no tempo de execução, focando em como disponibilizar os dados a serem processados na memória local ou aumentando a quantidade de dados disponíveis ao serem acessados da memória global.

CONCLUSÃO

Esta pesquisa apresentou um *framework* para coprojeto de hardware/software do módulo da dinâmica do modelo climático BRAMS, defendendo a tese de que esse *framework* é aplicável e auxilia na conversão e execução de código *Fortran* do BRAMS para *OpenCL*.

Como apresentado no capítulo 1, o objetivo geral desta tese foi projetar, implementar e avaliar um *framework* para coprojeto de hardware/software da dinâmica do modelo climático BRAMS, usando como plataforma de desenvolvimento e prototipação uma arquitetura heterogênea de computação composta por circuitos programáveis do tipo FPGA e processador Intel Xeon.

A pesquisa teve início com a condução de estudos sobre o BRAMS. Para isso, um tempo foi dedicado para analisar a versão completa do mesmo, verificando a viabilidade em trabalhar com todo o código. Após esse estudo optou-se por considerar o módulo da dinâmica em razão da complexidade do código total e indicação de especialistas do domínio. Durante esse processo, foi conduzida uma revisão de literatura com o objetivo de obter trabalhos relacionados ao domínio da pesquisa.

Tanto o estudo do BRAMS, quanto a revisão de literatura, serviram de base para a proposição do *framework* para coprojeto hardware/software. Para avaliar o *framework* proposto foram conduzidos dois estudos de caso exploratórios, utilizando o módulo da dinâmica do BRAMS. Os estudos tiveram como objetivo verificar a aplicabilidade do *framework* proposto.

Com os estudos de caso observou-se que o *framework* é promissor por ser aplicável na prática e pode contribuir com a atualização do código do BRAMS. O tempo investido na condução dos estudos foi muito grande devido a dificuldade em obter informações sobre o sistema, sobretudo sobre o código fonte e os processos envolvidos na previsão do tempo. Outra dificuldade foi a conversão do código *Fortran* para *OpenCL*, realizada de forma manual, inclusive a validação da conversão. Apesar disso, foi possível definir e utilizar o *framework*, determinando-o como aplicável na prática e auxiliar na atualização do código do BRAMS.

Dado que o BRAMS possui um código legado e necessita de atualizações, é válido migrar parte da aplicação para uma plataforma heterogênea de computação composta por CPU e FPGA. Apesar de não ter obtido ganho em performance, os *kernels* codificados neste trabalho para o módulo da dinâmica, como demonstrado nos estudos de caso, ganham em eficiência de energia. Em alguns casos, a diferença de desempenho para o tempo de execução ficou muito próxima da execução apenas em CPU. Isso representa um bom indicativo de que otimizações e a utilização de um FPGA mais poderoso computacionalmente - em número de elementos lógicos e interfaces de memória, como é o caso do Stratix 10 - pode vir a superar a performance da CPU.

Um outro ponto positivo a ser destacado é que o Intel *OpenCL* para FPGA mostrou-se uma linguagem de síntese de alto nível capaz de produzir um hardware competitivo com uma CPU moderna. Vale destacar que a versão do compilador utilizada não foi a mais recente já disponibilizada pela Intel, o que nesse caso poderia fornecer alguma melhora na geração de um hardware mais otimizado. A utilização dessa versão do compilador 16.0 ocorreu em virtude das interfaces de comunicação e camada de abstração fornecida pela Intel para a plataforma HARP, que até o momento não foi atualizada para a última versão do compilador *OpenCL* disponibilizado pela mesma. Por fim, o framework foi desenvolvido e avaliado para uma aplicação real responsável pela previsão do tempo no Brasil. Os códigos utilizados para a construção dos *kernels* são os mesmos em produção no CPTEC/INPE.

Por fim, com o desenvolvimento desta pesquisa comprovou-se que a tecnologia da computação reconfigurável está madura o suficiente para ser usada em sistemas voltados para aplicações de alto desempenho, expandindo assim o seu uso, pois tradicionalmente era aplicado em sistemas embarcados e com uso restrito em algumas áreas da computação, como, por exemplo, em *switches* e roteadores de redes de computadores.

7.1 Contribuições e limitações

Esta tese apresenta uma principal contribuição para a computação reconfigurável: um *framework* para coprojeto de hardware/software da dinâmica do modelo climático BRAMS, usando como plataforma de desenvolvimento e prototipação o FPGA.

Na contribuição foram definidos 19 *kernels* que visam verificar a possibilidade de acelerar o código da dinâmica usando *OpenCL* e FPGA. Estudos de caso exploratórios, utilizando o módulo da dinâmica do BRAMS foram conduzidos para avaliar o *framework* proposto. Outras contribuições do trabalho foram:

- Disponibilizar a primeira implementação e integração de *kernels OpenCL* do BRAMS para uma máquina heterogênea que combina no mesmo chip um processador Intel Xeon e um FPGA Arria 10;
- Disponibilizar um método em conjunto com um ambiente, no qual é possível converter

código *Fortran* do BRAMS para *OpenCL* e testar a compatibilidade do código gerado visando obter a mesma saída produzida pelo código *Fortran*;

- Integração do código *Fortran* do módulo da dinâmica do BRAMS com módulos de hardware escritos em *OpenCL*. Essa integração ocorreu com sucesso e foi possível demonstrar que outros módulos do BRAMS podem possuir blocos de hardware específicos com o objetivo de tentar acelerar a execução dos mesmos;
- Disponibilizar um ambiente em que é possível selecionar um trecho de código da dinâmica do BRAMS, instrumentar o código e executá-lo em conjunto com um hardware específico para o trecho selecionado.

Nesse sentido, com a pesquisa conduzida durante este doutorado foi possível responder as duas hipóteses elencadas no Capítulo 1:

- **1ª:** É possível esquematizar transformação de código *Fortran* para executar em um ambiente de computação heterogênea composto por CPU e FPGA?

Sim, é possível.

- **2ª:** É possível esquematizar transformação de código do BRAMS para um ambiente de computação heterogênea composto por CPU e FPGA com ganho de desempenho?

A pesquisa conduzida durante este doutorado e os estudos de caso conduzidos não comprovaram ganho de desempenho para o BRAMS utilizando um ambiente de computação heterogênea composto por CPU e FPGA. No entanto, foram comprovados ganhos relacionados a eficiência de energia.

Algumas limitações deste trabalho são listadas a seguir:

1. Os estudos de caso exploratórios foram conduzidos utilizando apenas o módulo da dinâmica do BRAMS em razão da complexidade do código total do sistema;
2. O sistema só pode ser executado com um processo. Isso ocorre em razão das bibliotecas disponibilizadas pelo Intel *OpenCL* não oferecerem suporte para a execuções com mais de um processo acessando o FPGA;
3. O tamanho do FPGA não possibilitou executar dois *kernels* em conjunto, pois o tamanho deles excedia a capacidade do FPGA quanto aos recursos lógicos disponíveis;
4. Não foi possível explorar otimizações em *OpenCL* para o hardware gerado devido ao grande tempo de compilação exigido e a máquina necessária para compilar os códigos gerados.

No ambiente do HARP é possível utilizar apenas um nó do cluster. Essa limitação impossibilitou executar a dinâmica do BRAMS com mais de um processo e utilizar mais de um FPGA. Outra limitação encontrada foi o fato do Intel OpenCL SDK ainda não permitir que mais de um processo possa utilizar o FPGA, mesmo que os processos não executassem no FPGA de forma simultânea, entrassem em uma fila ou utilizassem um token de permissão para executar. Para uma aplicação paralela como o BRAMS, seria necessário um cluster como o HARP, no entanto, com acesso e utilização de todos os nós e dispositivos FPGA em conjunto com os processadores Intel Xeon.

A utilização de um compilador e um BSP menos otimizados e desatualizados do ponto de vista de lançamento pela própria Intel representou outra limitação. O compilador utilizado neste trabalho foi o 16.0, porém a Intel já possui um compilador aoc na versão 18.1. Devido ao BSP disponibilizado para a Arria 10, presente no HARP, estar desatualizado não foi possível utilizar a última versão do compilador e, consequentemente, as melhorias.

Vale ressaltar que o HARP é um programa de pesquisa em desenvolvimento e para obter bons resultados faz-se necessário que a tecnologia disponibilizada pelo programa sofra atualizações à medida que as tecnologias dos FPGAs, compiladores e processadores da Intel evoluem.

7.2 Lições aprendidas

Durante o doutorado algumas atividades não foram executadas como inicialmente planejadas e alguns desafios não previstos precisaram ser superados.

No início da pesquisa, um dos maiores desafios do projeto foi entender o código do BRAMS que conta com 400.000 linhas de código. A documentação do código não é facilmente compreendida por um profissional fora da área de previsão do tempo. Foi necessário contato com especialistas e desenvolvedores do código para facilitar essa tarefa. Esse processo culminou na decisão de considerar o código do módulo da dinâmica, utilizado nos estudos de caso.

A execução remota dos testes foi outro desafio não previsto inicialmente. A máquina utilizada para execução dos testes está na Alemanha e é acessada remotamente. Inicialmente os testes eram feitos fora de um ambiente integrado de programação e execução. Esse problema foi resolvido ao utilizar a IDE IntelliJ IDEA, que possibilitou integrar essas atividades em um único sistema aumentando a produtividade.

A integração dos *kernels* em *OpenCL* com o código da dinâmica no ambiente do HARP foi uma atividade que exigiu esforço devido as dificuldades encontradas. Essa questão foi resolvida com o módulo desenvolvido para o framework, cujo objetivo foi conectar as bibliotecas *OpenCL* e da dinâmica do BRAMS com o código do host que faz uso do kernel gravado no FPGA.

7.3 Trabalhos futuros

Há diversas pesquisas que podem surgir a partir dos estudos conduzidos durante este doutorado e apresentados nesta tese. Um estudo mais imediato seria automatizar a conversão do código *Fortran* do BRAMS para *OpenCL*. Esse trabalho seria facilitado pelo estudo da estrutura de código do BRAMS identificada durante o desenvolvimento deste doutorado.

Outro trabalho futuro seria explorar os diferentes parâmetros de compilação do *OpenCL* e algumas técnicas de geração de hardware otimizado usando *OpenCL*. Esse trabalho contribuiria como auxílio na redução do tempo de execução dos *kernels* desenvolvidos neste trabalho de doutorado.

Outra atividade interessante seria o desenvolvimento de *kernels* para outros módulos do BRAMS com o intuito de explorar outras características do código que podem refletir em ganho de desempenho. No entanto, esse trabalho necessitaria do acompanhamento de um especialista do código para facilitar essa tarefa.

REFERÊNCIAS

ABBOTT, A.; ATHANAS, P. M.; TARMASER, A. Accelerating image filters using a custom computing machine. In: **Proceedings of SPIE - The International Society for Optical Engineering**. [S.l.: s.n.], 1995. v. 2607, p. 62–70. Citado na página [68](#).

ADIONO, T.; ARDYANTO, A.; AHMADI, N.; HAFIZH, I.; PUTRA, S. An SoC architecture for real-time noise cancellation system using variable speech PDF method. **International Journal of Electrical and Computer Engineering**, v. 5, n. 6, p. 1336–1346, 2015. Citado na página [72](#).

AHMAD, S.; BOPPANA, V.; GANUSOV, I.; KATHAIL, V.; RAJAGOPALAN, V.; WITTIG, R. A 16-nm multiprocessing system-on-chip field-programmable gate array platform. **IEEE Micro**, v. 36, n. 2, p. 48–62, Mar 2016. ISSN 0272-1732. Citado na página [51](#).

ALLAM, A.; DEABES, W. Electrical capacitance tomography digital processing platform (ECT-DPU). In: **IECON Proceedings (Industrial Electronics Conference)**. [S.l.: s.n.], 2016. p. 4767–4771. Citado na página [74](#).

ALVARADO, M.; ROSA, S. D. L. Design of a telecommunication, command and data handling system (TCDH) for a remote sensing mission. In: **Proceedings of the International Astronautical Congress, IAC**. [S.l.: s.n.], 2016. Citado na página [74](#).

ARNOLD, J. S5: the architecture and development flow of a software configurable processor. In: **Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on**. [S.l.: s.n.], 2005. p. 121–128. Citado na página [36](#).

ARNOLD, M.; FINK, S.; GROVE, D.; HIND, M.; SWEENEY, P. F. Adaptive optimization in the jalapeño JVM. In: **Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications**. [S.l.: s.n.], 2000. (OOPSLA '00), p. 47–65. ISBN 1-58113-200-X. Citado na página [44](#).

ARNOLD, M.; RYDER, B. G. A framework for reducing the cost of instrumented code. In: **Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation**. [S.l.: s.n.], 2001. (PLDI '01), p. 168–179. ISBN 1-58113-414-2. Citado na página [44](#).

AYSU, A.; YUCE, B.; SCHAUMONT, P. The future of real-time security: Latency-optimized lattice-based digital signatures. **ACM Transactions on Embedded Computing Systems**, v. 14, n. 3, 2015. Citado na página [72](#).

BACON, D.; RABBAH, R.; SHUKLA, S. FPGA programming for the masses. **Queue**, ACM, New York, NY, USA, v. 11, p. 40:40–40:52, 2013. Citado nas páginas [40](#) e [71](#).

BALA, V. **Low overhead path profiling**. [S.l.], 1996. Citado na página [46](#).

BALL, T.; LARUS, J. R. Efficient path profiling. In: **Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture**. [S.l.: s.n.], 1996. (MICRO 29), p. 46–57. ISBN 0-8186-7641-8. Citado na página [45](#).

BERNARDI, C. A. C.; MADALOSSO, O. M.; CHARÃO, A. S.; VELHO, H. F. C.; RUIZ, R. S. R. Explorando OpenCL na paralelização de uma sub-rotina do modelo ambiental CCATT-BRAMS. In: . [S.l.: s.n.], 2013. ESCOLA REGIONAL DE ALTO DESEMPENHO DO ESTADO DO RIO GRANDE DO SUL, 13. (ERAD-RS). Citado na página 80.

BITTWARE. **S5-PCIe-HQ - Altera Stratix V GX/GS Half-Length PCIe Board with Dual QSFP+, DDR3, and QDRII+ - User's Guide**. [S.l.], 2015. Disponível em: <ftp://ftp.bittware.com/documents/data_sheets/ds-s5phq.pdf>. Acesso em: Nov 11, 2018. Citado na página 91.

BOBDA, C. **Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications**. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2007. Citado na página 71.

_____. **Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications**. [S.l.]: Springer, Netherlands, 2010. ISBN 9789048175314. Citado nas páginas 47, 48 e 49.

BRAGE, J. P.; MADSEN, J. Codesign case study in computer graphics. In: **Hardware/Software Codesign - Proceedings of the International Workshop**. [S.l.: s.n.], 1994. p. 132–139. Citado na página 44.

CALDEIRA JERONIMO C. PENHA, L. B. R. F. J. A. M. N. R. F. P.; PEREIRA, F. M. Q. From Java to FPGA: an experience with the Intel HARP System. 2018. Acesso em: Nov 11, 2018. Citado na página 55.

CANIS, A.; CHOI, J.; ALDHAM, M.; ZHANG, V.; KAMMOONA, A.; CZAJKOWSKI, T.; BROWN, S.; ANDERSON, J. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. **Transactions on Embedded Computing Systems**, v. 13, n. 2, 2013. Citado na página 71.

CARDOSO, J.; HÜBNER, M. **Reconfigurable Computing: From FPGAs to Hardware/Software Codesign**. [S.l.]: Springer New York, 2011. (Electrical engineering). ISBN 9781461400615. Citado nas páginas 40, 41 e 42.

CARREÑO, E. D.; ROLOFF, E.; SANCHEZ, J. V.; NAVAUX, P. O. Performance analysis of a numerical weather prediction application in microsoft azure. In: . [S.l.: s.n.], 2015. WSPPD 2015 - XIII Workshop de Processamento Paralelo e Distribuído. Citado nas páginas 82 e 83.

CHILIMBI, T. M.; HIRZEL, M. Dynamic hot data stream prefetching for general-purpose programs. **SIGPLAN Not.**, v. 37, n. 5, p. 199–209, maio 2002. ISSN 0362-1340. Citado na página 44.

CMELIK, B. **SpixTools: Introduction and User's Manual**. [S.l.]: Sun Microsystems Laboratories, 1993. (SMLI technical report series). Citado na página 46.

CPTEC/INPE. Brazilian developments on the regional atmospheric modelling system. Model Description. 2018. Disponível em: <<http://brams.cptec.inpe.br/>>. Citado nas páginas 35, 62, 63, 64 e 99.

CRAY. 2016. Disponível em: <<http://www.cray.com/>>. Citado na página 73.

CUMMING, B.; OSUNA, C.; GYSI, T.; BIANCO, M.; LAPILLONNE, X.; FUHRER, O.; SCHULTHESS, T. C. A review of the challenges and results of refactoring the community climate code COSMO for hybrid cray HPC systems. **Proceedings of Cray User Group**, 2013. Citado nas páginas 34, 36, 73 e 74.

DURAN, R. E.; CHEN, D.; SARASWAT, R.; HALLMARK, A. A framework for comparing high performance computing technologies. **Int. J. Comput. Sci. Eng.**, Inderscience Publishers, Inderscience Publishers, Geneva, SWITZERLAND, v. 9, n. 1/2, p. 119–129, jan. 2014. ISSN 1742-7185. Disponível em: <<http://dx.doi.org/10.1504/IJCSE.2014.058704>>. Citado nas páginas 72 e 74.

EL-GHAZAWI, T.; EL-ARABY, E.; HUANG, M.; GAJ, K.; KINDRATENKO, V.; BUELL, D. The promise of high-performance reconfigurable computing. **Computer**, v. 41, n. 2, p. 69–76, Feb 2008. ISSN 0018-9162. Citado nas páginas 50 e 51.

ERNST, R.; HENKEL, J.; BENNER, T. Hardware-software cosynthesis for microcontrollers. **IEEE Des. Test**, v. 10, n. 4, p. 64–75, out. 1993. ISSN 0740-7475. Citado na página 67.

EVANGELISTA, L. E. S.; FAZENDA, A. L.; MELO, V. V. de. Paradigma dataflow para aplicação científica em GPGPU. In: . [S.l.: s.n.], 2014. WSCAD 2014 - XV Simpósio em Sistemas Computacionais de Alto Desempenho. Citado nas páginas 80 e 81.

FARRIMONDAND, B.; COLLINS, J.; SHARMA, A. APPRASE: Automatic parallelisation of fortran to run on an FPGA. 2009. Disponível em: <https://www.researchgate.net/publication/252055697_APPRASE_Automatic_parallelisation_of_Fortran_to_run_on_an_FPGA>. Citado na página 75.

FAZENDA, A. L.; ENARI, E. H.; RODRIGUES, L. F.; PANETTA, J. Towards production code effective portability among vector machines and microprocessor-based architectures. In: **2006 18th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'06)**. [S.l.: s.n.], 2006. p. 11–20. Citado na página 75.

FAZENDA, Á. L.; PANETTA, J.; RODRIGUES, L. F.; KATSURAYAMA, D. M.; MOTTA, L. F.; NAVAUX, P. O. A. Escalabilidade de aplicação operacional em ambiente massivamente paralelo. In: WSCAD-SSC 2009: SIMPÓSIO E SISTEMAS COMPUTACIONAIS, 10., 28-31 out., São Paulo. [S.l.], 2009. Acesso em: 25 abr. 2016. Citado nas páginas 77 e 78.

FERNANDES, A. d. A. **Paralelização do termo de reatividade química do modelo ambiental CCATT-BRAMS utilizando um solver baseado em estimação linear ótima**. 103 p. Dissertação (Mestrado) — Instituto Nacional de Pesquisas Espaciais (INPE), São José dos Campos, 2014-11-28 2014. Disponível em: <<http://urlib.net/sid.inpe.br/mtc-m21b/2014/12.01.15.57>>. Citado nas páginas 81 e 82.

FIRDAUS, A. F.; MERIBOUT, M. A new parallel VLSI architecture for real-time electrical capacitance tomography. **IEEE Transactions on Computers**, v. 65, n. 1, p. 30–41, 2016. Citado na página 69.

FOURNARIS, A.; DIMOPOULOS, C.; MOSCHOS, A.; KOUFOPAVLOU, O. Design and leakage assessment of side channel attack resistant binary edwards elliptic curve digital signature algorithm architectures. **Microprocessors and Microsystems**, v. 64, p. 73–87, 2019. Citado na página 74.

- FREIRE, J. L. M.; FREITAS, S. R. d.; COELHO, C. A. d. S. Calibração do modelo regional BRAMS para a previsão de eventos climáticos extremos. **Revista Brasileira de Meteorologia**, scielo, v. 30, p. 158 – 170, 06 2015. ISSN 0102-7786. Citado na página 35.
- FREITAS, S. R.; LONGO, K. M.; DIAS, M. A. F. S.; DIAS, P. L. S.; CHATFIELD, R.; PRINS, E.; ARTAXO, P.; GRELL, G. A.; RECUERO, F. S. Monitoring the transport of biomass burning emissions in south america. **Environmental Fluid Mechanics**, v. 5, n. 1, p. 135–167, Apr 2005. ISSN 1573-1510. Disponível em: <<https://doi.org/10.1007/s10652-005-0243-7>>. Citado na página 61.
- FREITAS, S. R.; LONGO, K. M.; DIAS, M. A. F. S.; CHATFIELD, R.; DIAS, P. S.; ARTAXO, P.; ANDREAE, M. O.; GRELL, G.; RODRIGUES, L. F.; FAZENDA, A.; PANETTA, J. 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**, v. 9, n. 8, p. 2843–2861, 2009. Disponível em: <<https://www.atmos-chem-phys.net/9/2843/2009/>>. Citado na página 61.
- FREITAS, S. R.; PANETTA, J.; LONGO, K. M.; RODRIGUES, L. F.; MOREIRA, D. S.; ROSÁRIO, N. E.; DIAS, P. L. S.; DIAS, M. A. F. S.; SOUZA, E. P.; FREITAS, E. D.; LONGO, M.; FRASSONI, A.; FAZENDA, A. L.; SILVA, C. M. Santos e; PAVANI, C. A. B.; EIRAS, D.; FRANÇA, D. A.; MASSARU, D.; SILVA, F. B.; SANTOS, F. C.; PEREIRA, G.; CAMPO-NGARA, G.; FERRADA, G. A.; VELHO, H. F. C.; MENEZES, I.; FREIRE, J. L.; ALONSO, M. F.; GÁCITA, M. S.; ZARZUR, M.; FONSECA, R. M.; LIMA, R. S.; SIQUEIRA, R. A.; BRAZ, R.; TOMITA, S.; OLIVEIRA, V.; MARTINS, L. D. The brazilian developments on the regional atmospheric modeling system (BRAMS 5.2): an integrated environmental model tuned for tropical areas. **Geoscientific Model Development**, v. 10, n. 1, p. 189–222, 2017. Disponível em: <<https://www.geosci-model-dev.net/10/189/2017/>>. Citado nas páginas 35, 61, 62 e 63.
- GAO, B.; REES, D. J. ASAP: an asynchronous array processor for hardware-software coprocessing and codesign. In: **International Conference on ASIC, Proceedings**. [S.l.: s.n.], 1996. p. 151–154. Citado na página 69.
- GOKHALE, M.; GRAHAM, P. S. **Reconfigurable Computing Accelerating Computation with Field-Programmable Gate Arrays**. 1st. ed. Dordrecht, Netherlands: Springer, 2005. Citado na página 48.
- GONZALEZ, C.; LOPEZ, S.; MOZOS, D.; SARMIENTO, R. FPGA implementation of the hysime algorithm for the determination of the number of endmembers in hyperspectral data. **IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing**, v. 8, n. 6, p. 2870–2883, 2015. Citado na página 69.
- _____. A novel fpga-based architecture for the estimation of the virtual dimensionality in remotely sensed hyperspectral images. **Journal of Real-Time Image Processing**, v. 15, n. 2, p. 297–308, 2018. Citado na página 69.
- GONZÁLEZ, C.; BERNABÉ, S.; MOZOS, D.; PLAZA, A. FPGA implementation of an algorithm for automatically detecting targets in remotely sensed hyperspectral images. **IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing**, v. 9, n. 9, p. 4334–4343, 2016. Citado na página 69.
- GOVIL, N.; CHOWDHURY, S. High performance and low cost implementation of fast fourier transform algorithm based on hardware software co-design. In: **IEEE TENSYP 2014 - 2014 IEEE Region 10 Symposium**. [S.l.: s.n.], 2014. p. 403–407. Citado na página 74.

GSCHWIND, M.; SALAPURA, V.; MAURER, D. FPGA prototyping of a RISC processor core for embedded applications. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 9, n. 2, p. 241–250, 2001. Citado na página 69.

GULATI, K.; KHATRI, S. P. **Hardware Acceleration of EDA Algorithms: Custom ICs, FPGAs and GPUs**. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2010. ISBN 1441909435, 9781441909435. Citado nas páginas 31, 32, 40, 41 e 43.

GUPTA, P. **Xeon+FPGA Platform for the Data Center**. [S.l.], 2015. Disponível em: <<https://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf>>. Acesso em: Apr 10, 2017. Citado nas páginas 53 e 54.

_____. Accelerating datacenter workloads. 2016. Disponível em: <<http://www.fpl2016.org/slides/Gupta--AcceleratingDatacenter20Workloads.pdf>>. Citado nas páginas 55 e 57.

GUPTA, R. K.; MICHELI, G. D. Hardware-software cosynthesis for digital systems. **IEEE Design Test of Computers**, v. 10, n. 3, p. 29–41, 1993. Citado na página 67.

HADOOP, A. 2016. Disponível em: <<http://hadoop.apache.org/>>. Citado na página 72.

HALFHILL, T. R. Intel's hybrid CPU-FPGA. 2016. Disponível em: <<http://www.embeddedintel.com/commentary.php?article=2143>>. Citado na página 32.

HARING, R.; OHMACHT, M.; FOX, T.; GSCHWIND, M.; SATTERFIELD, D.; SUGAVANAM, K.; COTEUS, P.; HEIDELBERGER, P.; BLUMRICH, M.; WISNIEWSKI, R.; GARA, A.; CHIU, G.-T.; BOYLE, P.; CHIST, N.; KIM, C. The IBM blue Gene/Q compute chip. **Micro, IEEE**, v. 32, n. 2, p. 48–60, March 2012. Citado na página 32.

HAUCK, S.; DEHON, A. **Reconfigurable Computing: the Theory and Practice of FPGA-Based Computation**. 1st. ed. Burlington, United States: Elsevier, 2008. Citado na página 48.

HPCL. Supercomputers. High Performance Computing LAB - The George Washington University. 2016. Disponível em: <<https://hpc12.hpcl.gwu.edu/index.php/supercomputers>>. Citado na página 50.

HPGC. Tsinghua high performance geo-computing (HPGC) group. 2015. Disponível em: <<http://www.thuhpgc.org/>>. Citado na página 33.

HSIUNG, P.-A.; SANTAMBROGIO, M. D.; HUANG, C.-H. **Reconfigurable System Design and Verification**. 1st. ed. London, New York: Taylor & Francis Group, CRC Press, 2009. Citado na página 48.

HUFFSTETLER, J. **Intel Processors and FPGAs—Better Together**. [S.l.], 2018. Disponível em: <<https://itpeernetwork.intel.com/intel-processors-fpga-better-together/>>. Acesso em: Nov 11, 2018. Citado nas páginas 54 e 56.

INC, M. T. 2016. Disponível em: <www.maxeler.com>. Citado na página 51.

INTEL. 2016. Disponível em: <<http://www.intel.com.br/content/www/br/pt/processors/xeon/xeon-processor-e5-family.html>>. Citado na página 72.

_____. 2016. Disponível em: <http://ark.intel.com/pt-br/products/77988/Intel-Atom-Processor-C2758-4M-Cache-2_40-GHz>. Citado na página 72.

_____. A global program to stimulate innovative research. 2018. Disponível em: <<https://software.intel.com/pt-br/hardware-accelerator-research-program>>. Citado na página 53.

_____. Intel fpga sdk for opencl pro edition: Programming guide. 2018. Disponível em: <<https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807965224.html#mwh1391807939093>>. Citado nas páginas 57, 58, 59, 60 e 102.

JAZIRI, I.; CHARAABI, L.; JELASSI, K. Dynamically reconfigurable DC motor control system based on co-design methodology. In: **7th International Conference on Sciences of Electronics, Technologies of Information and Telecommunications SETIT**. [S.l.: s.n.], 2017. p. 92–96. Citado na página 74.

JONES, D. H.; POWELL, A.; BOUGANIS, C.-S.; CHEUNG, P. Y. K. GPU versus FPGA for high productivity computing. In: **Proceedings of the 2010 International Conference on Field Programmable Logic and Applications**. Washington, DC, USA: IEEE Computer Society, 2010. (FPL '10, v. 1), p. 119–124. Citado na página 32.

KHRONOS. **The open standard for parallel programming of heterogeneous systems**. [S.l.], 2018. Disponível em: <<https://www.khronos.org/opencl/>>. Acesso em: Nov 11, 2018. Citado nas páginas 56 e 57.

KIMURA, S.; YUKISHITA, M.; ITOU, Y.; NAGOYA, A.; HIRAO, M.; WATANABE, K. Hardware/software codesign method for a general purpose reconfigurable co-processor. In: **Hardware/Software Codesign - Proceedings of the International Workshop**. [S.l.: s.n.], 1997. p. 147–151. Citado na página 69.

KISTLER, T.; FRANZ, M. Continuous program optimization: Design and evaluation. **IEEE Transactions on Computers**, v. 50, n. 6, p. 549–566, 2001. Citado na página 44.

LANCASTER, J. **Low-Impact Profiling of Streaming, Heterogeneous Applications**. Tese (Doutorado) — School of Engineering Applied Science, Washington University, 2011. Citado na página 44.

LAWRENCE, B. N.; REZNY, M.; BUDICH, R.; BAUER, P.; BEHRENS, J.; CARTER, M.; DECONINCK, W.; FORD, R.; MAYNARD, C.; MULLERWORTH, S.; OSUNA, C.; PORTER, A.; SERRADELL, K.; VALCKE, S.; WEDI, N.; WILSON, S. Crossing the chasm: how to develop weather and climate models for next generation computers? **Geoscientific Model Development**, v. 11, n. 5, p. 1799–1821, 2018. Disponível em: <<https://www.geosci-model-dev.net/11/1799/2018/>>. Citado nas páginas 33 e 34.

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**. New York, NY, USA: ACM, 2009. (ICAC-INDST '09, v. 1), p. 29–38. Citado na página 32.

MAGGIANI, L.; SALVADORI, C.; PETRACCA, M.; PAGANO, P.; SALETTI, R. Reconfigurable FPGA architecture for computer vision applications in smart camera networks. In: . [S.l.: s.n.], 2013. Citado na página 71.

MERTEN, M. C.; TRICK, A. R.; BARNES, R. D.; NYSTROM, E. M.; GEORGE, C. N.; GYLLENHAAL, J. C.; HWU, W. W. An architectural framework for runtime optimization. **IEEE Transactions on Computers**, v. 50, n. 6, p. 567–589, 2001. Citado na página 44.

METZ, C. To stay relevant, intel explores its biggest takeover ever. 2015. Disponível em: <<https://www.wired.com/2015/03/intel-exploring-biggest-takeover-ever/>>. Citado na página 53.

METZ, E.; LENCEVICIUS, R.; GONZALEZ, T. F. Performance data collection using a hybrid approach. In: **Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering**. [S.l.: s.n.], 2005. (ESEC/FSE-13), p. 126–135. ISBN 1-59593-014-0. Citado na página 45.

MILLER, B. P.; CALLAGHAN, M. D.; CARGILLE, J. M.; HOLLINGSWORTH, J. K.; IRVIN, R. B.; KARAVANIC, K. L.; KUNCHITHAPADAM, K.; NEWHALL, T. The paradyn parallel performance measurement tool. **Computer**, v. 28, n. 11, p. 37–46, 1995. Citado na página 44.

MISHRA, A.; GARG, K.; ASATI, A. R.; RAJU, K. S. Hardware software co-design using profiling and clustering. In: **International Conference on Communication, Information Computing Technology (ICCICT)**. [S.l.: s.n.], 2012. p. 1–4. Citado na página 44.

NANDY, S.; GAO, X.; FERRANTE, J. Tfp: Time-sensitive, flow-specific profiling at runtime. In: RAUCHWERGER, L. (Ed.). **Languages and Compilers for Parallel Computing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. p. 32–47. ISBN 978-3-540-24644-2. Citado na página 46.

NESHATPOUR, K.; MALIK, M.; GHODRAT, M.; HOMAYOUN, H. Accelerating big data analytics using FPGAs. In: **Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on**. [S.l.: s.n.], 2015. p. 164–164. Citado nas páginas 72 e 73.

NIEMANN, R.; MARWEDEL, P. An algorithm for hardware/software partitioning using mixed integer linear programming. **Design Automation for Embedded Systems**, v. 2, n. 2, p. 165–193, 1997. Citado na página 69.

PASC. Climate and atmospheric modelling. 2014. Disponível em: <<http://www.pasc-ch.org/networks/climate/>>. Citado na página 34.

PATEL, R.; RAJAWAT, A. A survey of embedded software profiling methodologies. **CoRR**, abs/1312.2949, 2013. Citado na página 46.

PIELKE, R.; COTTON, W.; WALKO, R.; TREMBACK, C.; LYONS, W.; GRASSO, L.; NICHOLLS, M.; MORAN, M.; WESLEY, D.; LEE, T.; COPELAND, J. A comprehensive meteorological modeling system-RAMS. **Meteorology and Atmospheric Physics**, Springer-Verlag, v. 49, n. 1-4, p. 69–91, 1992. Citado na página 35.

PRAKASH, S.; PARKER, A. C. SOS: Synthesis of application-specific heterogeneous multi-processor systems. **Journal of Parallel and Distributed Computing**, v. 16, n. 4, p. 338 – 351, 1992. ISSN 0743-7315. Citado na página 67.

RODRIGUES, E. R.; MADRUGA, F. L.; NAVAUX, P. O. A.; PANETTA, J. Multi-core aware process mapping and its impact on communication overhead of parallel applications. In: **Computers and Communications, 2009. ISCC 2009. IEEE Symposium on**. [S.l.: s.n.], 2009. p. 811–817. Citado nas páginas 76 e 77.

RODRIGUES, E. R.; NAVAUX, P. O. A.; PANETTA, J.; FAZENDA, A.; MENDES, C. L.; KALE, L. V. A comparative analysis of load balancing algorithms applied to a weather forecast model. In: **Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on**. [S.l.: s.n.], 2010. p. 71–78. Citado nas páginas 78 e 79.

SCHAUMONT, P. Hello pciex on terasic de2i-150. Hardware/Software Codesign with DE2i-150. 2013. Disponível em: <<http://rijndael.ece.vt.edu/de2i150/>>. Citado na página 88.

SCHAUMONT, P. R. **A Practical Introduction to Hardware/Software Codesign**. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2010. ISBN 1441959998, 9781441959997. Citado nas páginas 40, 41, 42 e 43.

SCHMIT, H.; THOMAS, D. Hidden Markov modeling and fuzzy controllers in FPGAs. In: **IEEE Symposium on FPGAs for Custom Computing Machines, Proceedings**. [S.l.: s.n.], 1995. p. 214–221. Citado na página 68.

SEBESTA, R. W. **Concepts of Programming Languages**. 10th. ed. USA: Addison-Wesley Publishing Company, 2013. ISBN 9780131395312. Citado na página 48.

SHANKAR, K. V. **Non-Intrusive Dynamic Application Profiling And Error Detection For Embedded Systems**. Dissertação (Mestrado) — Graduate Collegem, The University of Arizona, 2010. Citado na página 45.

SOUZA, C. A. O. D.; PEREIRA, E. D. S.; MARQUES, E. A hardware/software codesign for the chemical reactivity of BRAMS. In: **2017 Euromicro Conference on Digital System Design (DSD)**. [S.l.: s.n.], 2017. p. 70–77. Citado na página 82.

STRETCH, S. I. Stretch ide. 2018. Disponível em: <http://www.stretchinc.com/_files/Stretch_IDE_Development_Tool.pdf>. Citado nas páginas 69, 97 e 98.

TERASIC. DE2i-150 FPGA development kit. 2015. Disponível em: <<http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=163&No=529&PartNo=1>>. Citado nas páginas 87 e 88.

THOMA, Y.; DASSATTI, A.; MOLLA, D.; PETRAGLIO, E. FPGA-GPU communicating through PCIe. **Microprocess. Microsyst.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 39, n. 7, p. 565–575, out. 2015. ISSN 0141-9331. Disponível em: <<http://dx.doi.org/10.1016/j.micpro.2015.02.005>>. Citado nas páginas 32 e 70.

TONG, G.; KHALID, J. .; MOHAMMED. Profiling tools for FPGA-based embedded systems: Survey and quantitative comparison. **Journal of Computers**, v. 3, p. 1–14, 2008. Citado nas páginas 44 e 46.

TRIPOLI, G.; COTTON, W. The colorado state university three-dimensional cloud/mesoscale model - 1982. part i: general theoretical framework and sensitivity experiments. **Journal de Recherches Atmospheriques**, v. 16, p. 185–219, 01 1982. Citado na página 62.

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**. New York, NY, USA: ACM, 2010. (FPGA '10, v. 1), p. 115–124. ISBN 978-1-60558-911-4. Disponível em: <<http://doi.acm.org/10.1145/1723112.1723134>>. Citado nas páginas 31 e 32.

- VANDERBAUWHEDE, W.; BENKRID, K. **High-Performance Computing Using FPGAs**. Springer New York, 2013. (SpringerLink : Bücher). ISBN 9781461417910. Disponível em: <https://books.google.com.br/books?id=uRh_AAAAQBAJ>. Citado nas páginas 51 e 52.
- WANG, C.; LI, X.; ZHOU, X.; NEDJAH, N.; WANG, A. Codem: Software/hardware code-sign for embedded multicore systems supporting hardware services. **International Journal of Electronics**, v. 102, n. 1, p. 32–47, 2015. Citado nas páginas 44 e 46.
- WEINHARDT, M.; LANG, B.; THIESING, F. M.; KRIEGER, A.; KINDER, T. SAccO: An implementation platform for scalable FPGA accelerators. **Microprocessors and Microsystems**, v. 39, n. 7, p. 543 – 552, 2015. ISSN 0141-9331. Citado nas páginas 51 e 71.
- WENDLING, M.; ROSENSTIEL, W. Hardware environment for prototyping and partitioning based on multiple FPGAs. In: **European Design Automation Conference - Proceedings**. [S.l.: s.n.], 1994. p. 77–82. Citado na página 68.
- WESTON, S.; SPOONER, J.; RACANIERE, S.; MENCER, O. Rapid computation of value and risk for derivatives portfolios. **Concurr. Comput. : Pract. Exper.**, John Wiley and Sons Ltd., Chichester, UK, v. 24, n. 8, p. 880–894, jun. 2012. ISSN 1532-0626. Citado na página 33.
- WOLF, W. A decade of hardware/software codesign. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 36, n. 4, p. 38–43, 2003. Citado nas páginas 40, 67, 68 e 69.
- XILINX. 2016. Disponível em: <<http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html>>. Citado na página 72.
- XU, Z. Measuring green it in society. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 45, n. 5, p. 83–85, maio 2012. ISSN 0018-9162. Disponível em: <<http://dx.doi.org/10.1109/MC.2012.176>>. Citado na página 32.
- YALI, M. P.; SCHAUMONT, P. AES co-processor on terasic DE2i-150. Hardware/Software Codesign with DE2i-150. 2013. Disponível em: <<http://rijndael.ece.vt.edu/de2i150/>>. Citado nas páginas 89 e 90.
- YUAN, Z.; LIU, Y.; LI, J.; HU, J.; XUE, C.; YANG, H. CP-FPGA: Energy-efficient nonvolatile FPGA with offline/online checkpointing optimization. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 25, n. 7, p. 2153–2163, 2017. Citado na página 72.
- ZOHOURI, H. R. High performance computing with FPGAs and OpenCL. **CoRR**, abs/1810.09773, 2018. Citado nas páginas 48, 49, 50, 58 e 59.

VISÃO DOS KERNELS GERADA DURANTE A COMPILAÇÃO DO CÓDIGO *OpenCL*

Em virtude do tamanho dos outros *kernels* pertencentes ao grupo *Grad* é apresentado apenas o sistema do *kernel 5*.

Em virtude do tamanho dos *kernels* 1 e 2 pertencentes ao grupo *acustic* é apresentado o sistema dos *kernels* 3, 4 e 5.

Figura 49 – Visão do *kernel* 1 da turbulência em nível de *loops* e operações de *load* e *store*

Fonte: Elaborada pelo autor.

Figura 50 – Visão do *kernel 2* da turbulência em nível de *loops* e operações de *load* e *store*

Fonte: Elaborada pelo autor.

Figura 51 – Visão do kernel 3 da turbulência em nível de *loops* e operações de *load* e *store*

Fonte: Elaborada pelo autor.

Figura 52 – Visão do *kernel 4* da turbulência em nível de *loops* e operações de *load* e *store*

Fonte: Elaborada pelo autor.

Figura 53 – Visão do *kernel* 1 do grupo *Advect* em nível de *loops* e operações de *load* e *store*

Fonte: Elaborada pelo autor.

Figura 54 – Visão do *kernel 2* do grupo *Advect* em nível de *loops* e operações de *load* e *store*

Fonte: Elaborada pelo autor.

Figura 55 – Visão do *kernel 3* do grupo *Advect* em nível de *loops* e operações de *load* e *store*

Fonte: Elaborada pelo autor.

Figura 56 – Visão do *kernel* 4 do grupo *Advect* em nível de *loops* e operações de *load* e *store*

Fonte: Elaborada pelo autor.

Figura 57 – Visão do *kernel 5* do grupo *Advect* em nível de *loops* e operações de *load* e *store*

Fonte: Elaborada pelo autor.

Figura 58 – Visão do kernel 5 do grupo Grad em nível de loops e operações de load e store

Fonte: Elaborada pelo autor.

Figura 59 – Visão do *kernel 3* do grupo *acustic* em nível de *loops* e operações de *load* e *store*

Fonte: Elaborada pelo autor.

Figura 60 – Visão do *kernel* 4 do grupo *acustic* em nível de *loops* e operações de *load* e *store*

Figura 61 – Visão do *kernel 5* do grupo *acustic* em nível de *loops* e operações de *load* e *store*

Fonte: Elaborada pelo autor.

CÓDIGOS-FONTE EM FORTRAN DOS KERNELS SELECIONADOS E DESENVOLVIDOS EM OPENCL

Os códigos apresentados correspondem aos kernels do módulo da turbulência presente na dinâmica do BRAMS. Eles foram selecionados a partir de um profile linha a linha (*Line-by-line Profiling*, opção 1 do gprof) do código da dinâmica do BRAMS. Eles pertencem a sub-rotina *diffsclr* contida no arquivo *turb_diff.f90*.

Turbulência - Kernel 1

```

1  do j=ja,jz
2      do i=ia,iz
3
4          vt3df(2,i,j) = vt3dc(2,i,j) * vt2db(i,j)
5
6          do k=3,mzp-1
7              vt3df(k,i,j) = (vt3dc(k,i,j) - vt3dj(k,i,j) * vt3df(k
-1,i,j)) * vt3dm(k,i,j)
8          enddo
9
10         do k=mzp-2,2,-1
11             vt3df(k,i,j) = vt3df(k,i,j) - vt3dl(k,i,j) * vt3df(k
+1,i,j)
12         enddo
13
14     enddo
15 enddo

```

Turbulência - Kernel 2

```

1 dtliti = 1.0/dtlt
2
3  if (jd==1) then
4      do j=ja,jz
5          do i=ia,iz
6              do k=2,mzp-1
7                  sct(k,i,j) = sct(k,i,j) - scp(k,i,j)*dtliti - &
8                      (vt3da(k,i,j) + vt3db(k,i,j))*dn03i(k,i,j)
9              enddo
10             enddo
11         enddo
12     else
13         do j=ja,jz
14             do i=ia,iz
15                 do k=2,mzp-1
16                     sct(k,i,j) = sct(k,i,j) - scp(k,i,j)*dtliti - &
17                         vt3da(k,i,j)*dn03i(k,i,j)
18                 enddo
19             enddo
20         enddo
21     endif
22
23     do j=ja,jz
24         do i=ia,iz
25             do k=2,mzp-1
26                 sct(k,i,j) = sct(k,i,j) + vt3df(k,i,j)*dtliti
27             enddo
28         enddo
29     enddo

```

Turbulência - Kernel 3

```

1 do j=ja,jz
2     do i=ia,iz
3         do k=3,mzp-1
4             vt3dm(k,i,j) = 1./(vt3do(k,i,j) - vt3dj(k,i,j)*
5                 vt3dl(k-1,i,j))
6             vt3dl(k,i,j) = vt3dk(k,i,j)*vt3dm(k,i,j)
7         enddo
8     enddo
9 enddo

```

Turbulência - Kernel 4

```

1  do j = ja,jz
2      do i = ia,iz
3
4          cjp1(2,i,j) = cip1(2,i,j) / ci(2,i,j)
5          rhs(2,i,j) = rhs(2,i,j) / ci(2,i,j)
6
7          do k = 3,kz
8              cj(k,i,j) = ci(k,i,j) - cimzp(k,i,j) * cjp1(k-1,i,j)
9              cji = 1. / cj(k,i,j)
10             cjp1(k,i,j) = cip1(k,i,j) * cji
11             rhs(k,i,j) = (rhs(k,i,j) - cimzp(k,i,j) * rhs(k-1,i,j)
12         ) * cji
13         enddo
14
15         cj(kz,i,j) = rhs(kz,i,j)
16
17         do k = kz-1,2,-1
18             cj(k,i,j) = rhs(k,i,j) - cjp1(k,i,j) * cj(k+1,i,j)
19         enddo
20     enddo

```

Os códigos apresentados a seguir correspondem aos *kernels* do grupo **Grad**. Eles pertencem ao arquivo *rgrad.f90* do conjunto de códigos do BRAMS.

Grad - Kernel 1

```

1  SUBROUTINE GRADXT(mzp,mxp,my,ia,iz,ja,jz  &
2      ,OPTYP,VC3DA,VC3DB,RTGE,RTGC  &
3      ,DX,DZ,FMAPI,FMAP,FQ,HQ,LEV,JD)
4
5      implicit none
6
7      integer, INTENT(IN) :: mzp,mxp,my,ia,iz,ja,jz,jd
8
9      real, INTENT(IN) :: VC3DA(mzp,mxp,my)  &
10         , RTGE(mxp,my)  &
11         , RTGC(mxp,my)  &
12         , DX(mxp,my)  &
13         , FMAP(mxp,my)  &
14         , FMAPI(mxp,my)  &
15         , DZ(*)  &

```

```

16          , FQ (mxp , myp)          &
17          , HQ (*)
18
19  real , INTENT (INOUT) :: VC3DB (mzp , mxp , myp)
20  character (len=*) , INTENT (IN) :: OPTYP , LEV
21
22  integer :: i , j , k
23  real :: vc1da (mzp)
24  real :: hq4 (mzp)
25
26  IF (OPTYP .EQ. 'GRADNT') THEN
27      DO J=ja , jz
28          DO I=ia , iz
29              DO K=1 , mzp
30                  VC3DB (K , I , J) = (VC3DA (K , I+1 , J) * RTGE (I+1 , J)    &
31                      - VC3DA (K , I , J) * RTGE (I , J))    &
32                      * DX (I , J) / RTGC (I , J)
33              ENDDO
34          ENDDO
35      ENDDO
36  ELSE
37      DO J=ja , jz
38          DO I=ia , iz
39              DO K=1 , mzp
40                  VC3DB (K , I , J) = (VC3DA (K , I+1 , J) * RTGE (I+1 , J)    &
41                      * FMAPI (I+1 , J)    &
42                      - VC3DA (K , I , J) * RTGE (I , J)    &
43                      * FMAPI (I , J))    &
44                      * DX (I , J) / RTGC (I , J) * FMAP (I , J)
45              ENDDO
46          ENDDO
47      ENDDO
48  ENDIF
49
50  IF (OPTYP .NE. 'DIVSTR') THEN
51      IF (LEV .EQ. 'W') THEN
52          DO K=1 , mzp
53              HQ4 (K) = 0.25 * HQ (K)
54          ENDDO
55      ELSE
56          DO K=2 , mzp
57              HQ4 (K) = 0.25 * HQ (K-1)

```

```

58      ENDDO
59  ENDIF
60
61  do j=ja,jz
62      do i=ia,iz
63          do k=2,mzp
64              vc1da(k)=hq4(k)*(vc3da(k,i,j)+vc3da(k-1,i,j)  &
65                  +vc3da(k,i+1,j)+vc3da(k-1,i+1,j))
66          enddo
67
68          IF(OPTYP .NE. 'GRADNT') VC1DA(2)=0.
69
70          do k=2,mzp-1
71              vc3db(k,i,j)=vc3db(k,i,j)  &
72                  +fq(i,j)*dz(k)*(vc1da(k+1)-vc1da(k))
73          enddo
74      enddo
75  enddo
76  do j=ja,jz
77      do i=ia,iz
78          vc3db(1,i,j)=vc3db(2,i,j)
79      enddo
80  enddo
81  if (lev.eq.'W') then
82      do j=ja,jz
83          do i=ia,iz
84              vc3db(mzp-1,i,j)=vc3db(mzp-2,i,j)
85          enddo
86      enddo
87  else if(lev.eq.'T') then
88      do j=ja,jz
89          do i=ia,iz
90              vc3db(mzp,i,j)=vc3db(mzp-1,i,j)
91          enddo
92      enddo
93  end if
94
95  ENDIF
96 END SUBROUTINE GRADXT

```

Grad - Kernel 2

```

1  subroutine gradxu(mzp,mpx,my,ia,iz,ja,jz  &

```

```

2      , optyp, vc3da, vc3db, rtge, rtgc  &
3      , dx, dz, fmap, fapi, fq, hq, lev, jd)
4
5  implicit none
6
7  integer, INTENT(IN) :: mzp  &
8                        , mxp  &
9                        , myp  &
10                     , ia  &
11                     , iz  &
12                     , ja  &
13                     , jz  &
14                     , jd
15
16  real, INTENT(IN) :: vc3da(mzp, mxp, myp)  &
17                        , rtge(mxp, myp)      &
18                        , rtgc(mxp, myp)      &
19                        , dx(mxp, myp)        &
20                        , fmap(mxp, myp)      &
21                        , fapi(mxp, myp)      &
22                        , dz(mzp)             &
23                        , fq(mxp, myp)        &
24                        , hq(*)
25
26  real, INTENT(OUT) :: vc3db(mzp, mxp, myp)
27  character(len=*), INTENT(IN) :: optyp, lev
28
29  integer :: i, j, k
30  real :: vc1da(mzp)
31  real :: hq4(mzp)
32
33  IF (OPTYP.EQ. 'GRADNT') THEN
34      DO J=ja, jz
35          DO I=ia, iz
36              DO K=1, mzp
37                  VC3DB(K, I, J) = (VC3DA(K, I, J)*RTGE(I, J)  &
38                      - VC3DA(K, I-1, J)*RTGE(I-1, J))  &
39                      *DX(I, J)/RTGC(I, J)
40              ENDDO
41          ENDDO
42      ENDDO
43  ELSE

```

```

44     DO J=ja,jz
45         DO I=ia,iz
46             DO K=1,mzp
47                 VC3DB(K,I,J)=(VC3DA(K,I,J)*RTGE(I,J)  &
48                     *FMAPI(I,J)  &
49                     -VC3DA(K,I-1,J)*RTGE(I-1,J)  &
50                     *FMAPI(I-1,J))  &
51                     *DX(I,J)/RTGC(I,J)*FMAP(I,J)
52             ENDDO
53         ENDDO
54     ENDDO
55 ENDIF
56
57 IF(OPTYP.NE.'DIVSTR') THEN
58     IF(LEV.EQ.'W') THEN
59         DO K=1,mzp
60             HQ4(K)=0.25*HQ(K)
61         ENDDO
62     ELSE
63         DO K=2,mzp
64             HQ4(K)=0.25*HQ(K-1)
65         ENDDO
66     ENDIF
67
68     do j=ja,jz
69         do i=ia,iz
70             do k=2,mzp
71                 vc1da(k)=hq4(k)*(vc3da(k,i,j)+vc3da(k-1,i,j)  &
72                     +vc3da(k,i-1,j)+vc3da(k-1,i-1,j))
73             enddo
74
75             IF(OPTYP.NE.'GRADNT') VC1DA(2)=0.
76
77             do k=2,mzp-1
78                 vc3db(k,i,j)=vc3db(k,i,j)  &
79                     +fq(i,j)*dz(k)*(vc1da(k+1)-vc1da(k))
80             enddo
81         enddo
82     enddo
83     do j=ja,jz
84         do i=ia,iz
85             vc3db(1,i,j)=vc3db(2,i,j)

```

```

86         enddo
87     enddo
88     if(lev.eq.'W')then
89         do j=ja,jz
90             do i=ia,iz
91                 vc3db(mzp-1,i,j)=vc3db(mzp-2,i,j)
92             end do
93         end do
94     else if(lev.eq.'T')then
95         do j=ja,jz
96             do i=ia,iz
97                 vc3db(mzp,i,j)=vc3db(mzp-1,i,j)
98             end do
99         end do
100     end if
101 ENDIF
102 END subroutine gradxu

```

Grad - Kernel 3

```

1 SUBROUTINE GRADYT(mzp,mxp,my,ia,iz,ja,jz  &
2     ,OPTYP,VC3DA,VC3DB,RTGE,RTGC  &
3     ,DY,DZ,FMAPI,FMAP,FQ,HQ,LEV,JD)
4
5 implicit none
6
7 integer, INTENT(IN) :: mzp  &
8                        , mxp  &
9                        , myp  &
10                       , ia  &
11                       , iz  &
12                       , ja  &
13                       , jz  &
14                       , jd
15
16 real, INTENT(IN) :: VC3DA(mzp,mxp,my)  &
17                        , RTGE(mxp,my)  &
18                        , RTGC(mxp,my)  &
19                        , DY(mxp,my)  &
20                        , FMAP(mxp,my)  &
21                        , FMAPI(mxp,my)  &
22                        , DZ(*)  &
23                        , FQ(mxp,my)  &

```



```

24         , HQ(*)
25
26     real, INTENT(INOUT) :: VC3DB(mzp, mxp, myp)
27     character(len=*), INTENT(IN) :: OPTYP, LEV
28
29     integer :: i, j, k
30     real :: vc1da(mzp)
31     real :: hq4(mzp)
32
33     IF (OPTYP.EQ.'GRADNT') THEN
34         DO J=ja, jz
35             DO I=ia, iz
36                 DO K=1, mzp
37                     VC3DB(K, I, J) = (VC3DA(K, I, J+jd)*RTGE(I, J+jd) &
38                     -VC3DA(K, I, J)*RTGE(I, J)) &
39                     *DY(I, J)/RTGC(I, J)
40                 ENDDO
41             ENDDO
42         ENDDO
43     ELSE
44         DO J=ja, jz
45             DO I=ia, iz
46                 DO K=1, mzp
47                     VC3DB(K, I, J) = (VC3DA(K, I, J+jd)*RTGE(I, J+jd) &
48                     *FMAPI(I, J+jd) &
49                     -VC3DA(K, I, J)*RTGE(I, J) &
50                     *FMAPI(I, J)) &
51                     *DY(I, J)/RTGC(I, J)*FMAP(I, J)
52                 ENDDO
53             ENDDO
54         ENDDO
55     ENDIF
56
57     IF (OPTYP.NE.'DIVSTR') THEN
58         IF (LEV.EQ.'W') THEN
59             DO K=1, mzp
60                 HQ4(K) = 0.25*HQ(K)
61             ENDDO
62         ELSE
63             DO K=2, mzp
64                 HQ4(K) = 0.25*HQ(K-1)
65             ENDDO

```

```

66     ENDIF
67
68     do j=ja,jz
69         do i=ia,iz
70             do k=2,mzp
71                 vc1da(k)=hq4(k)*(vc3da(k,i,j)+vc3da(k-1,i,j)  &
72                     +vc3da(k,i,j+jd)+vc3da(k-1,i,j+jd))
73             enddo
74
75             IF(OPTYP .NE. 'GRADNT') VC1DA(2)=0.
76
77             do k=2,mzp-1
78                 vc3db(k,i,j)=vc3db(k,i,j)  &
79                     +fq(i,j)*dz(k)*(vc1da(k+1)-vc1da(k))
80             enddo
81         enddo
82     enddo
83     do j=ja,jz
84         do i=ia,iz
85             vc3db(1,i,j)=vc3db(2,i,j)
86         enddo
87     enddo
88     if (lev.eq.'W') then
89         do j=ja,jz
90             do i=ia,iz
91                 vc3db(mzp-1,i,j)=vc3db(mzp-2,i,j)
92             enddo
93         enddo
94     else if (lev.eq.'T') then
95         do j=ja,jz
96             do i=ia,iz
97                 vc3db(mzp,i,j)=vc3db(mzp-1,i,j)
98             enddo
99         enddo
100     end if
101
102     ENDIF
103 END SUBROUTINE GRADYT

```

Grad - Kernel 4

```

1 SUBROUTINE GRADYV(mzp,mxp,my,ia,iz,ja,jz  &
2     ,OPTYP,VC3DA,VC3DB,RTGE,RTGC  &

```

```

3      ,DY,DZ,FMAPI,FMAP,FQ,HQ,LEV,JD)
4
5  implicit none
6
7  integer, INTENT(IN) :: mzp    &
8                          , mxp    &
9                          , myp    &
10                     , ia    &
11                     , iz    &
12                     , ja    &
13                     , jz    &
14                     , jd
15
16  real, INTENT(IN)  :: VC3DA(mzp,mxp,myp)    &
17                          , RTGE(mxp,myp)      &
18                          , RTGC(mxp,myp)      &
19                          , DY(mxp,myp)      &
20                          , FMAP(mxp,myp)      &
21                          , FMAPI(mxp,myp)      &
22                          , DZ(*)              &
23                          , FQ(mxp,myp)      &
24                          , HQ(*)
25
26  real, INTENT(INOUT) :: VC3DB(mzp,mxp,myp)
27
28  character(len=*) :: OPTYP,LEV
29  integer :: i,j,k
30  real :: vc1da(mzp)
31  real :: hq4(mzp)
32
33  IF(OPTYP.EQ.'GRADNT') THEN
34      DO J=ja,jz
35          DO I=ia,iz
36              DO K=1,mzp
37                  VC3DB(K,I,J)=(VC3DA(K,I,J)*RTGE(I,J)    &
38                      -VC3DA(K,I,J-jd)*RTGE(I,J-jd))    &
39                      *DY(I,J)/RTGC(I,J)
40              ENDDO
41          ENDDO
42      ENDDO
43  ELSE
44      DO J=ja,jz

```

```

45      DO I=ia,iz
46          DO K=1,mzp
47              VC3DB(K,I,J)=(VC3DA(K,I,J)*RTGE(I,J)  &
48                  *FMAPI(I,J)  &
49                  -VC3DA(K,I,J-jd)*RTGE(I,J-jd)  &
50                  *FMAPI(I,J-jd))  &
51                  *DY(I,J)/RTGC(I,J)*FMAP(I,J)
52          ENDDO
53      ENDDO
54  ENDDO
55  ENDIF
56
57  IF(OPTYP.NE.'DIVSTR') THEN
58      IF(LEV.EQ.'W') THEN
59          DO K=1,mzp
60              HQ4(K)=0.25*HQ(K)
61          ENDDO
62      ELSE
63          DO K=2,mzp
64              HQ4(K)=0.25*HQ(K-1)
65          ENDDO
66      ENDIF
67
68      do j=ja,jz
69          do i=ia,iz
70              do k=2,mzp
71                  vc1da(k)=hq4(k)*(vc3da(k,i,j)+vc3da(k-1,i,j)  &
72                      +vc3da(k,i,j-jd)+vc3da(k-1,i,j-jd))
73              enddo
74
75              IF(OPTYP.NE.'GRADNT') VC1DA(2)=0.
76
77              do k=2,mzp-1
78                  vc3db(k,i,j)=vc3db(k,i,j)  &
79                      +fq(i,j)*dz(k)*(vc1da(k+1)-vc1da(k))
80              enddo
81          enddo
82      enddo
83      do j=ja,jz
84          do i=ia,iz
85              vc3db(1,i,j)=vc3db(2,i,j)
86          enddo

```

```

87     enddo
88     if (lev.eq.'W') then
89         do j=ja,jz
90             do i=ia,iz
91                 vc3db(mzp-1,i,j)=vc3db(mzp-2,i,j)
92             enddo
93         enddo
94     else if (lev.eq.'T') then
95         do j=ja,jz
96             do i=ia,iz
97                 vc3db(mzp,i,j)=vc3db(mzp-1,i,j)
98             enddo
99         enddo
100     end if
101 ENDIF
102 END SUBROUTINE GRADYV

```

Grad - Kernel 5

```

1 SUBROUTINE GRADZT(mzp,mxp,my,ia,iz,ja,jz,VC3DA,VC3DB,RTGC,DZ)
2
3     implicit none
4     integer :: mzp,mxp,my,ia,iz,ja,jz
5
6     real, INTENT(IN) :: VC3DA(mzp,mxp,my) &
7                        , RTGC(mxp,my) &
8                        , DZ(*)
9
10    real, INTENT(INOUT) :: VC3DB(mzp,mxp,my)
11
12    integer :: i,j,k
13
14    DO J=ja,jz
15        DO I=ia,iz
16            DO K=1,mzp-1
17                VC3DB(K,I,J)=(VC3DA(K+1,I,J)-VC3DA(K,I,J))*DZ(K) &
18                            /RTGC(I,J)
19            ENDDO
20        ENDDO
21    ENDDO
22 END SUBROUTINE GRADZT

```

Os códigos apresentados a seguir correspondem aos *kernels* do grupo **Advect**, módulo

que calcula a advecção do BRAMS. Eles pertencem ao arquivo *radvc.f90* do conjunto de códigos do BRAMS.

Advect - Kernel 1

```

1  subroutine fa_xc(mzp,mxp,my,ia,iz,ja,jz, &
2      scp,scr1,vt3da,vt3dd,vt3dg,vt3dh,vt3di)
3
4      implicit none
5      integer, intent(in) :: mzp
6      integer, intent(in) :: mxp
7      integer, intent(in) :: myp
8      integer, intent(in) :: ia
9      integer, intent(in) :: iz
10     integer, intent(in) :: ja
11     integer, intent(in) :: jz
12     real, intent(in) :: scp(mzp,mxp,my)
13     real, intent(inout) :: scr1(mzp,mxp,my)
14     real, intent(in) :: vt3da(mzp,mxp,my)
15     real, intent(in) :: vt3dd(mzp,mxp,my)
16     real, intent(out) :: vt3dg(mzp,mxp,my)
17     real, intent(in) :: vt3dh(mzp,mxp,my)
18     real, intent(in) :: vt3di(mzp,mxp,my)
19
20     integer :: i,j,k
21     real :: dfact
22
23     dfact = .5
24     do j = 1,my
25         do i = 1, iz
26
27             do k = 2,mzp-1
28                 vt3dg(k,i,j) = vt3da(k,i,j) &
29                     * (vt3di(1,i,j) * scr1(k,i,j) &
30                     + vt3di(2,i,j) * scr1(k,i+1,j) &
31                     + vt3dd(k,i,j) * (scr1(k,i,j) - scr1(k,i+1,j)))
32             )
33
34             do k = 2,mzp-1
35                 if (vt3da(k,i,j) .gt. 0.) then
36                     if (vt3dg(k,i,j) * vt3dh(k,i,j) .gt. &
37                         dfact * scr1(k,i,j)) then

```

```

38         vt3dg(k,i,j) = vt3da(k,i,j) * scr1(k,i,j)
39     end if
40     elseif (vt3da(k,i,j) .lt. 0.) then
41         if (-vt3dg(k,i,j) * vt3dh(k,i+1,j) .gt. &
42             dfact * scr1(k,i+1,j)) then
43             vt3dg(k,i,j) = vt3da(k,i,j) * scr1(k,i+1,j)
44         end if
45     end if
46 end do
47 end do
48 end do
49
50 do j = 1, myp
51     do i = ia, iz
52         do k = 2, mzp-1
53             scr1(k,i,j) = scr1(k,i,j) &
54                 + vt3dh(k,i,j) * (vt3dg(k,i-1,j) - vt3dg(k,i,j))
55                 &
56                 + scp(k,i,j) * (vt3da(k,i,j) - vt3da(k,i-1,j)))
57         end do
58     end do
59 end do
end subroutine fa_xc

```

Advect - Kernel 2

```

1  subroutine fa_yc(mzp, mxp, myp, ia, iz, ja, jz, &
2      scp, scr1, vt3db, vt3de, vt3dg, vt3dj, vt3di, jdim)
3
4      implicit none
5      integer, intent(in) :: mzp
6      integer, intent(in) :: mxp
7      integer, intent(in) :: myp
8      integer, intent(in) :: ia
9      integer, intent(in) :: iz
10     integer, intent(in) :: ja
11     integer, intent(in) :: jz
12
13     real, intent(in) :: scp(mzp, mxp, myp)
14     real, intent(inout) :: scr1(mzp, mxp, myp)
15     real, intent(in) :: vt3db(mzp, mxp, myp)
16     real, intent(in) :: vt3de(mzp, mxp, myp)
17     real, intent(out) :: vt3dg(mzp, mxp, myp)

```

```

18  real, intent(in) :: vt3dj(mzp,mxp,my)
19  real, intent(in) :: vt3di(mzp,mxp,my)
20  integer, intent(in) :: jdim
21
22  integer :: i,j,k
23  real :: dfact
24
25  dfact = .5
26  do j = 1,jz
27      do i = ia,iz
28
29          do k = 2,mzp-1
30              vt3dg(k,i,j) = vt3db(k,i,j) &
31                  * (vt3di(3,i,j) * scr1(k,i,j) &
32                  + vt3di(4,i,j) * scr1(k,i,j+jdim) &
33                  + vt3de(k,i,j) * (scr1(k,i,j) - scr1(k,i,j+
jdim)))
34          end do
35
36          do k = 2,mzp-1
37              if (vt3db(k,i,j) .gt. 0.) then
38                  if (vt3dg(k,i,j) * vt3dj(k,i,j) .gt. &
39                      dfact * scr1(k,i,j)) then
40                      vt3dg(k,i,j) = vt3db(k,i,j) * scr1(k,i,j)
41                  end if
42              elseif (vt3db(k,i,j) .lt. 0.) then
43                  if (-vt3dg(k,i,j) * vt3dj(k,i,j+jdim) .gt. &
44                      dfact * scr1(k,i,j+jdim)) then
45                      vt3dg(k,i,j) = vt3db(k,i,j) * scr1(k,i,j+jdim)
46                  end if
47              end if
48          end do
49      end do
50  end do
51
52  do j = ja,jz
53      do i = ia,iz
54          do k = 2,mzp-1
55              scr1(k,i,j) = scr1(k,i,j) &
56              + vt3dj(k,i,j) * (vt3dg(k,i,j-jdim) - vt3dg(k,i
,j) &
57              + scp(k,i,j) * (vt3db(k,i,j) - vt3db(k,i,j-jdim

```



```

)))
58         end do
59     end do
60 end do
61 end subroutine fa_yc

```

Advect - Kernel 3

```

1 subroutine fa_zc(mzp, mxp, myp, ia, iz, ja, jz, &
2     scp, scr1, vt3dc, vt3df, vt3dg, vt3dk, vctr1, vctr2)
3
4     implicit none
5     integer, intent(in) :: mzp
6     integer, intent(in) :: mxp
7     integer, intent(in) :: myp
8     integer, intent(in) :: ia
9     integer, intent(in) :: iz
10    integer, intent(in) :: ja
11    integer, intent(in) :: jz
12    real, intent(in) :: scp(mzp, mxp, myp)
13    real, intent(inout) :: scr1(mzp, mxp, myp)
14    real, intent(in) :: vt3dc(mzp, mxp, myp)
15    real, intent(in) :: vt3df(mzp, mxp, myp)
16    real, intent(out) :: vt3dg(mzp, mxp, myp)
17    real, intent(in) :: vt3dk(mzp, mxp, myp)
18    real, intent(in) :: vctr1(mzp)
19    real, intent(in) :: vctr2(mzp)
20
21    integer :: i, j, k
22    real :: dfact
23
24    dfact = .5
25    do j = ja, jz
26        do i = ia, iz
27
28            do k = 1, mzp-1
29                vt3dg(k, i, j) = vt3dc(k, i, j) &
30                    * (vctr1(k) * scr1(k, i, j) &
31                    + vctr2(k) * scr1(k+1, i, j) &
32                    + vt3df(k, i, j) * (scr1(k, i, j) - scr1(k+1, i, j)))
33            )
34        end do

```

```

35      do k = 1,mzp-1
36          if (vt3dc(k,i,j) .gt. 0.) then
37              if (vt3dg(k,i,j) * vt3dk(k,i,j) .gt. &
38                  dfact * scr1(k,i,j)) then
39                  vt3dg(k,i,j) = vt3dc(k,i,j) * scr1(k,i,j)
40              end if
41          elseif (vt3dc(k,i,j) .lt. 0.) then
42              if (-vt3dg(k,i,j) * vt3dk(k+1,i,j) .gt. &
43                  dfact * scr1(k+1,i,j)) then
44                  vt3dg(k,i,j) = vt3dc(k,i,j) * scr1(k+1,i,j)
45              end if
46          end if
47      end do
48  end do
49 end do
50
51 do j = ja,jz
52     do i = ia,iz
53         do k = 2,mzp-1
54             scr1(k,i,j) = scr1(k,i,j) &
55                 + vt3dk(k,i,j) * (vt3dg(k-1,i,j) - vt3dg(k,i,j)
56 &
57                 + scp(k,i,j) * (vt3dc(k,i,j) - vt3dc(k-1,i,j)))
58         end do
59     end do
60 end do
end subroutine fa_zc

```

Advect - Kernel 4

```

1  subroutine advtndc(mzp,mxp,my,ia,iz,ja,jz,scp,sca,sct,dtl)
2
3  implicit none
4  integer, intent(in) :: mzp
5  integer, intent(in) :: mxp
6  integer, intent(in) :: my
7  integer, intent(in) :: ia
8  integer, intent(in) :: iz
9  integer, intent(in) :: ja
10 integer, intent(in) :: jz
11 real, intent(in) :: scp(mzp,mxp,my)
12 real, intent(in) :: sca(mzp,mxp,my)
13 real, intent(inout) :: sct(mzp,mxp,my)

```

```

14     real, intent(in) :: dtl
15
16     integer :: i,j,k
17     real :: dtli
18
19     dtli = 1. / dtl
20     do j = ja,jz
21         do i = ia,iz
22             do k = 2,mzp-1
23                 sct(k,i,j) = sct(k,i,j) + (sca(k,i,j)-scp(k,i,j)) *
dtli
24             end do
25         end do
26     end do
27 end subroutine advtndc

```

Advect - Kernel 5

```

1 subroutine fa_preptc(mzp,mxp,my,ia,iz,ja,jz,vt3da,vt3db,vt3dc,
   vt3dd,vt3de,vt3df, &
2     vt3dh,vt3di,vt3dj,vt3dk,vctr1, vctr2,dn0,dn0u,dn0v, &
3     rtgt,rtgu,rtgv,fmapt,fmapui,fmapvi,f13t,f23t, &
4     dxu,dyv,dxt,dyt,hw4, dzm, dzt, zm, zt)
5
6     implicit none
7     integer, intent(in) :: mzp
8     integer, intent(in) :: mxp
9     integer, intent(in) :: myp
10    integer, intent(in) :: ia,iz,ja,jz
11    real, intent(in) :: hw4(mzp), dzm(mzp), dzt(mzp), zm(mzp), zt
(mzp)
12    real, intent(inout) :: vt3da(mzp,mxp,my)
13    real, intent(inout) :: vt3db(mzp,mxp,my)
14    real, intent(inout) :: vt3dc(mzp,mxp,my)
15    real, intent(out) :: vt3dd(mzp,mxp,my)
16    real, intent(out) :: vt3de(mzp,mxp,my)
17    real, intent(out) :: vt3df(mzp,mxp,my)
18    real, intent(out) :: vt3dh(mzp,mxp,my)
19    real, intent(out) :: vt3di(mzp,mxp,my)
20    real, intent(out) :: vt3dj(mzp,mxp,my)
21    real, intent(out) :: vt3dk(mzp,mxp,my)
22    real, intent(in) :: dn0(mzp,mxp,my)
23    real, intent(in) :: dn0u(mzp,mxp,my)

```

```

24   real, intent(in) :: dn0v(mzp, mxp, myp)
25   real, intent(in) :: rtgt(mxp, myp)
26   real, intent(in) :: rtgu(mxp, myp)
27   real, intent(in) :: rtgv(mxp, myp)
28   real, intent(in) :: fmapt(mxp, myp)
29   real, intent(in) :: fmapui(mxp, myp)
30   real, intent(in) :: fmapvi(mxp, myp)
31   real, intent(in) :: f13t(mxp, myp)
32   real, intent(in) :: f23t(mxp, myp)
33   real, intent(in) :: dxu(mxp, myp)
34   real, intent(in) :: dyv(mxp, myp)
35   real, intent(in) :: dxt(mxp, myp)
36   real, intent(in) :: dyt(mxp, myp)
37   real, intent(out) :: vctr1(mzp)
38   real, intent(out) :: vctr2(mxp)
39
40   integer :: j, i, k, im, ip, jm, jp
41   real :: c1, c2, c3, c4, rtgti
42   real :: vctr3(mzp)
43
44   do j=1, myp
45       jm = max(1, j-1)
46       jp = min(myp, j+1)
47       do i=1, mxp
48           im = max(1, i-1)
49           ip = min(mxp, i+1)
50           rtgti = 1. / rtgt(i, j)
51           c1 = .5 * dxu(i, j)
52           c2 = .5 * dyv(i, j)
53           c3 = dxt(i, j) * fmapt(i, j) * rtgti
54           c4 = dyt(i, j) * fmapt(i, j) * rtgti
55
56           do k = 1, mzp-1
57               vt3dc(k, i, j) = ((vt3da(k, i, j) + vt3da(k+1, i, j) &
58                   + vt3da(k, im, j) + vt3da(k+1, im, j)) * f13t(i, j)
59                   &
60                   + (vt3db(k, i, j) + vt3db(k+1, i, j) + vt3db(k, i, jm
61                   + vt3db(k+1, i, jm)) * f23t(i, j)) * hw4(k) &
62                   + vt3dc(k, i, j) * rtgti
63               vt3dd(k, i, j) = c1 * vt3da(k, i, j)
64               vt3de(k, i, j) = c2 * vt3db(k, i, j)

```

```

64         vt3df(k,i,j) = .5 * vt3dc(k,i,j) * dzm(k)
65         vctr3(k) = 1. / dn0(k,i,j)
66         vt3dh(k,i,j) = c3 * vctr3(k)
67         vt3dj(k,i,j) = c4 * vctr3(k)
68         vt3dk(k,i,j) = dzt(k) * vctr3(k)
69     end do
70
71     vt3di(1,i,j) = .5
72     vt3di(2,i,j) = .5
73     vt3di(3,i,j) = .5
74     vt3di(4,i,j) = .5
75 end do
76 end do
77
78 do k = 1,mzp-1
79     vctr1(k) = (zt(k+1) - zm(k)) * dzm(k)
80     vctr2(k) = (zm(k) - zt(k)) * dzm(k)
81 end do
82
83 do j=1,my p
84     do i=1,mx p
85         c1 = fmapui(i,j) * rtgu(i,j)
86         c2 = fmapvi(i,j) * rtgv(i,j)
87         do k = 1,mzp-1
88             vt3da(k,i,j) = vt3da(k,i,j) * c1 * dn0u(k,i,j)
89             vt3db(k,i,j) = vt3db(k,i,j) * c2 * dn0v(k,i,j)
90             vt3dc(k,i,j) = vt3dc(k,i,j) * .5 &
91                 * (dn0(k,i,j) + dn0(k+1,i,j))
92         end do
93     end do
94 end do
95 end subroutine fa_preptc

```

Os códigos apresentados a seguir correspondem aos *kernels* do grupo **Acoustic**. Eles pertencem ao arquivo *raco.f90* do conjunto de códigos do BRAMS.

Acoustic - Kernel 1

```

1 subroutine coefz(mzp,mxp,my p,ia,iz,ja,jz,acoc,acof,acog,dn0,pi0,
   th0,rtgt,a1da2,amoe,amof,acoaa,&
2         impl,dts,sspct,dzm,dzt,nzp,nz,rgas,cv)
3
4     implicit none

```

```

5      integer, intent(in) :: mzp
6      integer, intent(in) :: mxp
7      integer, intent(in) :: myp
8      integer, intent(in) :: ia
9      integer, intent(in) :: iz
10     integer, intent(in) :: ja
11     integer, intent(in) :: jz
12     real, intent(out) :: acoc(mzp,mxp,myp)
13     real, intent(out) :: acof(mzp,mxp,myp)
14     real, intent(out) :: acog(mzp,mxp,myp)
15     real, intent(in) :: dn0(mzp,mxp,myp)
16     real, intent(in) :: pi0(mzp,mxp,myp)
17     real, intent(in) :: th0(mzp,mxp,myp)
18     real, intent(in) :: rtgt(mxp,myp)
19     real, intent(out) :: a1da2
20     real, intent(out) :: amoe(mzp,mxp,myp)
21     real, intent(out) :: amof(mzp,mxp,myp)
22     real, intent(out) :: acoaa(mzp,mxp,myp)
23     integer, intent(in) :: impl,nzp,nz
24     real, intent(in) :: dts,sspct
25     real, dimension(mzp), intent(in) :: dzm,dzt
26     real, intent(in) :: rgas,cv
27
28     integer :: i,j,k
29     real :: dt2al2,rdto2cv,dt2al2r,rdtr
30     real :: acobb(mzp)
31     real :: acocc(mzp)
32     real, dimension(mzp) :: vctr11,vctr12
33
34     if (impl .eq. 1) then
35         dt2al2 = dts * .75
36         a1da2 = 1. / 3.
37     else
38         dt2al2 = dts
39         a1da2 = 1.
40     endif
41     rdto2cv = sspct ** 2 * rgas * dts / (2.0 * cv)
42
43     do j = ja,jz
44         do i = ia,iz
45
46             dt2al2r = .5 * dt2al2 / rtgt(i,j)

```

```

47         do k = 1,mzp-1
48             acoc(k,i,j) = dt2a12r * dzm(k) * (th0(k,i,j) + th0(k
+1,i,j))
49         enddo
50
51         rdtr = rdto2cv / rtgt(i,j)
52         do k = 2,mzp
53             vctr12(k) = dn0(k,i,j) * th0(k,i,j)
54             vctr11(k) = rdtr * pi0(k,i,j) * dzt(k) / vctr12(k)
55         enddo
56         vctr12(1) = dn0(1,i,j) * th0(1,i,j)
57         do k = 2,mzp-1
58             acof(k,i,j) = -vctr11(k) * (vctr12(k) + vctr12(k+1))
59             acog(k,i,j) = vctr11(k) * (vctr12(k) + vctr12(k-1))
60         enddo
61
62         acog(mzp,i,j) = vctr11(nzp) * (vctr12(nzp) + vctr12(nz
)
63         acof(mzp,i,j) = -vctr11(nzp) * (vctr12(nzp) + vctr12(nz
))
64
65         do k = 2,mzp-1
66             acoaa(k,i,j) = acoc(k,i,j) * acog(k,i,j)
67             acobb(k) = acoc(k,i,j) * (acof(k,i,j) - acog(k+1,i,j
)) - 1.
68             acocc(k) = -acoc(k,i,j) * acof(k+1,i,j)
69         enddo
70
71         acobb(1) = -1.
72         acocc(1) = 0.
73         acoaa(mzp,i,j) = 0.
74         acobb(mzp) = -1.
75         amof(1,i,j) = acobb(1)
76         amoe(1,i,j) = acocc(1) / amof(1,i,j)
77
78         do k = 2,mzp
79             amof(k,i,j) = acobb(k) - acoaa(k,i,j) * amoe(k-1,i,j
)
80             amoe(k,i,j) = acocc(k) / amof(k,i,j)
81         enddo
82
83     enddo

```

```

84     enddo
85 end subroutine coefz

```

Acoustic - Kernel 2

```

1 subroutine prdctp1(mzp, mxp, myp, ia, iz, ja, jz, pp, up, vp, pi0, dn0, th0,
   pt, f13t, f23t, rtgt, rtgu, rtgv, &
2     heatfx1, fmapui, fmapvi, dxt, dyt, fmapt, sspct, jdim, hw4,
   dzt, dts, roc)
3
4     implicit none
5     integer, intent(in) :: mzp
6     integer, intent(in) :: mxp
7     integer, intent(in) :: myp
8     integer, intent(in) :: ia
9     integer, intent(in) :: iz
10    integer, intent(in) :: ja
11    integer, intent(in) :: jz
12    real, intent(inout) :: pp(mzp, mxp, myp)
13    real, intent(in) :: up(mzp, mxp, myp)
14    real, intent(in) :: vp(mzp, mxp, myp)
15    real, intent(in) :: pi0(mzp, mxp, myp)
16    real, intent(in) :: dn0(mzp, mxp, myp)
17    real, intent(in) :: th0(mzp, mxp, myp)
18    real, intent(in) :: pt(mzp, mxp, myp)
19    real, intent(in) :: f13t(mxp, myp)
20    real, intent(in) :: f23t(mxp, myp)
21    real, intent(in) :: rtgt(mxp, myp)
22    real, intent(in) :: rtgu(mxp, myp)
23    real, intent(in) :: rtgv(mxp, myp)
24    real, intent(out) :: heatfx1(mxp, myp)
25    real, intent(in) :: fmapui(mxp, myp)
26    real, intent(in) :: fmapvi(mxp, myp)
27    real, intent(in) :: dxt(mxp, myp)
28    real, intent(in) :: dyt(mxp, myp)
29    real, intent(in) :: fmapt(mxp, myp)
30    integer, intent(in) :: jdim
31    real, dimension(mzp), intent(in) :: hw4, dzt
32    real, intent(in) :: sspct, dts, roc
33
34    integer :: i, j, k
35    real :: rocvpct
36    real :: heatdv(mzp, mxp, myp)

```



```

37     real :: heatfx(mzp,mxp,my)
38
39     heatdv = 0.
40     rocvpct = rocv * sspct ** 2
41
42     do j = ja,jz
43         do i = ia,iz
44             do k = 1,mzp
45                 heatfx(k,i,j) = ((up(k,i,j) + up(k,i-1,j)) * f13t(i,
j) &
46                                     + (vp(k,i,j) + vp(k,i,j-jdim)) * f23t(i,j) &
47                                     ) * dn0(k,i,j) * th0(k,i,j)
48             enddo
49         enddo
50     enddo
51     do j = ja,jz
52         do i = ia,iz
53             do k = 1,mzp-1
54                 heatfx(k,i,j) = (heatfx(k,i,j) + heatfx(k+1,i,j)) *
hw4(k)
55             enddo
56             heatfx1(i,j) = heatfx(1,i,j) / (.5 * (dn0(1,i,j) * th0
(1,i,j) &
57                                     + dn0(2,i,j) * th0(2,i,j)))
58         enddo
59     enddo
60
61     do j = ja,jz
62         do i = ia,iz
63             do k = 2,mzp-1
64                 heatdv(k,i,j) = (heatfx(k,i,j) - heatfx(k-1,i,j)) *
dzt(k)
65             enddo
66         enddo
67     enddo
68     do j = 1,my
69         do i = 1,mxp
70             do k = 1,mzp
71                 heatfx(k,i,j) = dn0(k,i,j) * th0(k,i,j)
72             enddo
73         enddo
74     enddo

```

```

75     do j = ja,jz
76         do i = ia,iz
77             do k = 2,mzp-1
78
79                 heatdv(k,i,j) = -rocvpct * pi0(k,i,j) / heatfx(k,i,j
80             ) &
81                 * (heatdv(k,i,j) + fmapt(i,j) &
82                 * ((up(k,i,j) * rtgu(i,j) * fmapui(i,j) &
83                 * (heatfx(k,i,j) + heatfx(k,i+1,j)) &
84                 - up(k,i-1,j) * rtgu(i-1,j) * fmapui(i-1,j) &
85                 * (heatfx(k,i,j) + heatfx(k,i-1,j))) * dxt(i,j)
86
87             * .5 &
88
89                 + (vp(k,i,j) * rtgv(i,j) * fmapvi(i,j) &
90                 * (heatfx(k,i,j) + heatfx(k,i,j+jdim)) &
91                 - vp(k,i,j-jdim) * rtgv(i,j-jdim) &
92                 * fmapvi(i,j-jdim) &
93                 * (heatfx(k,i,j) + heatfx(k,i,j-jdim))) &
94                 * dyt(i,j) * .5) / rtgt(i,j))
95
96             enddo
97         enddo
98     enddo
99
100     do j = ja,jz
101         do i = ia,iz
102             do k = 1,mzp
103                 pp(k,i,j) = pp(k,i,j) + (pt(k,i,j) + heatdv(k,i,j))
104             * dts
105             enddo
106         enddo
107     enddo
108 end subroutine prdctp1

```

Acoustic - Kernel 3

```

1 subroutine prdctp2(mzp,mxp,my,ia,iz,ja,jz,pp,wp,acof,acog,nstbot
2     ,dzm)
3
4     implicit none
5     integer, intent(in) :: mzp
6     integer, intent(in) :: mxp
7     integer, intent(in) :: my
8     integer, intent(in) :: ia

```

```

8   integer, intent(in) :: iz
9   integer, intent(in) :: ja
10  integer, intent(in) :: jz
11  real, intent(inout) :: pp(mzp,mxp,my)
12  real, intent(in) :: wp(mzp,mxp,my)
13  real, intent(in) :: acof(mzp,mxp,my)
14  real, intent(in) :: acog(mzp,mxp,my)
15  integer, intent(in) :: nstbot
16  real, dimension(mzp), intent(in) :: dzm
17
18  integer :: i,j,k
19  real :: dzmr
20
21  do j = ja,jz
22      do i = ia,iz
23          do k = 2,mzp-1
24              pp(k,i,j) = pp(k,i,j) &
25                  + (wp(k,i,j) * acof(k,i,j) + wp(k-1,i,j) * acog
(k,i,j))
26          enddo
27      enddo
28  enddo
29
30  if (nstbot .eq. 1) then
31      dzmr = dzm(2) / dzm(1)
32      do i = 1,mxp
33          do j = 1,my
34              pp(1,i,j) = pp(2,i,j) + (pp(2,i,j) - pp(3,i,j)) *
dzmr
35          enddo
36      enddo
37  end if
38  end subroutine prdctp2

```

Acoustic - Kernel 4

```

1  subroutine prdctw2(mzp,mxp,my,ia,iz,ja,jz,wp,pp,acoc,amof,amog,
    acoaa,rtgt,heatfx1,&
2      nstbot,nsttop,impl,nz)
3
4      implicit none
5      integer, intent(in) :: mzp
6      integer, intent(in) :: mxp

```

```

7      integer, intent(in) :: myp
8      integer, intent(in) :: ia
9      integer, intent(in) :: iz
10     integer, intent(in) :: ja
11     integer, intent(in) :: jz
12     real, intent(inout) :: wp(mzp, mxp, myp)
13     real, intent(in) :: pp(mzp, mxp, myp)
14     real, intent(in) :: acoc(mzp, mxp, myp)
15     real, intent(in) :: amof(mzp, mxp, myp)
16     real, intent(out) :: amog(mzp, mxp, myp)
17     real, intent(in) :: acoaa(mzp, mxp, myp)
18     real, intent(in) :: rtgt(mxp, myp)
19     real, intent(in) :: heatfx1(mxp, myp)
20     integer, intent(in) :: nstbot, nsttop, impl, nz
21
22     integer :: i, j, k
23
24     if (nstbot .eq. 1) then
25         do j = ja, jz
26             do i = ia, iz
27                 wp(1, i, j) = -heatfx1(i, j) * rtgt(i, j)
28             enddo
29         enddo
30     endif
31
32     if (nsttop .eq. 1) then
33         do j = ja, jz
34             do i = ia, iz
35                 wp(nz, i, j) = 0.
36             enddo
37         enddo
38     endif
39
40     if (impl .eq. 1) then
41         do j = ja, jz
42             do i = ia, iz
43                 do k = 2, mzp-2
44                     wp(k, i, j) = wp(k, i, j) - (pp(k+1, i, j) - pp(k, i, j))
45                     * acoc(k, i, j)
46                 enddo
47             enddo
48         enddo
49     endif

```

```

48
49     do j = ja,jz
50         do i = ia,iz
51             amog(1,i,j) = -wp(1,i,j) / amof(1,i,j)
52         enddo
53     do k = 2,mzp-2
54         do i = ia,iz
55             amog(k,i,j) = (-wp(k,i,j) - acoaa(k,i,j) * amog(k
-1,i,j)) &
56                                     / amof(k,i,j)
57         enddo
58     enddo
59 enddo
60 endif
61 end subroutine prdctw2

```

Acoustic - Kernel 5

```

1 subroutine prdctw3(mzp,mxp,my,ia,iz,ja,jz,wp,amog,amoe,impl)
2
3     implicit none
4     integer, intent(in) :: mzp
5     integer, intent(in) :: mxp
6     integer, intent(in) :: myp
7     integer, intent(in) :: ia
8     integer, intent(in) :: iz
9     integer, intent(in) :: ja
10    integer, intent(in) :: jz
11    real, intent(inout) :: wp(mzp,mxp,my)
12    real, intent(in) :: amog(mzp,mxp,my)
13    real, intent(in) :: amoe(mzp,mxp,my)
14    integer, intent(in) :: impl
15
16    integer :: i,j,k
17    if (impl .eq. 1) then
18        do k = mzp-2,2,-1
19            do j = ja,jz
20                do i = ia,iz
21                    wp(k,i,j) = amog(k,i,j) - amoe(k,i,j) * wp(k+1,i,
j)
22                enddo
23            enddo
24        enddo

```

```
25     endif  
26 end subroutine prdctw3
```

DINÂMICA DO BRAMS VERSÃO 1

Dinâmica do BRAMS Versão 1

Jairo Panetta

Março de 2017

Finalidade

Gerar versão mínima do BRAMS 5.2 que facilite seu uso em pesquisas visando arquiteturas de computadores e formas de programação, mantendo intacta a estrutura de paralelismo. Para tanto, foi removida do BRAMS 5.2 toda a física, mantendo integralmente a dinâmica.

Obtendo Fontes e Dados

Obtenha os arquivos `DinamicaBRAMS_V1.tgz` e `DadosMinimosDinamica.tgz` no endereço <http://ftp-supercomputacao.inpe.br/public/jairo.panetta/>. Esses arquivos contêm os programas fonte e os dados necessários para executar esta versão, respectivamente. Além desses arquivos, o endereço publica cópia deste documento.

Obtenha os arquivos assim que possível, pois a política da supercomputação do CPTEC remove arquivos nessa área pública após 30 dias de sua instalação. Caso os arquivos já tenham sido removidos, solicite-os por e-mail para jairo.panetta@gmail.com.

Créditos e Limitações Conhecidas

O programa fonte é fruto do penoso trabalho de alguns anos da Simone Shizue Tomita Lima, que louvo.

Iniciei este trabalho no interior da rotina *timestep*. Removi as invocações das rotinas da física. Removi os arquivos correspondentes, sempre que possível. Sobraram todo o código das fases *MAKESFC*, *MAKEVFILE* e apenas a dinâmica da fase *INITIAL*. Preparei dois *RAMSIN* (60x60 e 100x100) que invocam apenas a dinâmica. Voltei ao fonte da *timestep*, removendo opções da dinâmica que considere obsoletas (como condições de fronteira circulares e *BRAMS global*) e opções com erros de codificação (como “shaved eta”). Culpa minha se a dinâmica restante não atender à opção que lhe interessa. Trabalhei ligeiramente na inicialização, eliminando rotinas desnecessárias. Entreguei esse fonte à Simone.

Simone trabalhou na árvore de chamadas da rotina *timestep*. Transformou todas as variáveis globais (via “*use association*”) em argumentos dos respectivos procedimentos, carregando esses argumentos até a *timestep*. Retirou todos os “*save*” implícitos ou explícitos. Essas modificações geraram código “*thread-safe*”.

Em seguida, unificou os limites dos laços que percorrem o subdomínio horizontal do processo MPI dentro da *timestep*, eliminando limites 2 e *mxp-1* ou *myp-1*, substituindo-os por *ia*, *iz*, *ja*, *jz*. Os limites 2, *mxp-1* e *myp-1* impediam alterar a largura da “*ghost zone*”, fixa em um por essas práticas. Neste código, é possível definir “*ghost zone*” de largura superior a um, o que é necessário para inserir advecção de alta ordem. Exceção são os procedimentos em que o método numérico exige “*ghost zone*” unitária, como a advecção original.

Resultou código com mais de 110.000 linhas. Certamente, ainda há muito a remover e limpar, mas é código muito mais dócil que o código original do BRAMS. Pretendo continuar trabalhando no processo de limpeza.

Dentre as limitações conhecidas, há o uso de HDF5, que me parece desnecessário. É sabido que o código ainda não tem reprodutibilidade binária. É sabido que faz inicializações desnecessárias. Mas compila e executa corretamente nas plataformas tupa (Cray no CPTEC sob compilador pgi), Santos Dumont (BULL no LNCC sob compilador intel) e phi03 (nó Intel no NCC/UNESP com Xeon acelerado por Xeon Phi KNC, quer no Xeon quer no Phi, sobre compilador intel).

Instalação

Copie os dois arquivos para sua área de trabalho.

Desempacote o arquivo `DinamicaBRAMS_V1.tgz` (com `"tar -xzf DinamicaBRAMS_V1.tgz"`) criando os diretórios *build*, *runDinamica* e *src*. O diretório *src* contém os arquivos fonte. O diretório *build* contém arquivos `Makefile` para compilação. O diretório *runDinamica* contém os arquivos necessários à execução.

Mova o arquivo `DadosMinimosDinamica.tgz` para o diretório *runDinamica*. Desempacote esse arquivo (com `"tar -xzf DadosMinimosDinamica.tgz"`), criando o diretório *DadosMinimosDinamica*, que contém apenas os dados necessários para a previsão de tempo por 24 horas para 27/01/2014 em região específica do Brasil.

Compilação

O diretório *build/bin* contém os arquivos necessários à compilação com `"make"` (`Makefile` e arquivos incluídos). Nesse diretório há arquivos `"include"` nomeados `include.mk.opt.<plataforma>-<compilador>` com os nomes dos compiladores e as chaves de compilação na respectiva plataforma. Esses arquivos `"include"` foram validados nas plataformas citadas (NCC/UNESP, tupa/CPTEC e Santos Dumont/LNCC). Busque o arquivo `"include"` adequado a sua plataforma.

Edite o `Makefile`. Apenda a linha `OPT=` com o nome do arquivo `"include"` escolhido, a partir de `"opt"`. Por exemplo, `"OPT=opt.tupa-pgi"` para compilar com pgi no tupa do CPTEC. Ou `"OPT=opt.sdumont-intel"` para compilar com intel no Santos Dumont do LNCC. Também há `"includes"` para alguns compiladores em qualquer plataforma (como `"include.mk.opt.intel"`) que não foram testados e precisam ser ajustados à sua plataforma.

Infelizmente, compilar esta versão da dinâmica exige compilador que inclua a biblioteca HDF5 paralela. Para ajustar o ambiente antes da compilação, abra o `"include"` na plataforma escolhida e siga as instruções na linha `"#environment"`. Obviamente, não há instruções para os `"includes"` em plataforma genérica.

Após ajustar o ambiente, vá ao diretório *build/bin* e execute `"make"` para compilar o código fonte. Se quiser limpar totalmente a compilação, execute `"make clean"`.

Compilações corretas geram o executável *build/dinamica.exe*.

Execução

Para executar a dinâmica, vá para o diretório *runDinamica*. Crie os arquivos *dinamica.exe* e *RAMSIN* utilizando links simbólicos: `"ln -s ../build/dinamica.exe`

`dinamica.exe` e `"ln -s RAMSIN_60x60 RAMSIN"`. Se quiser, substitua no segundo comando `"60x60"` por `"100x100"` para executar o *RAMSIN* do domínio horizontal maior (100x100).

Edite o arquivo *RAMSIN* e escolha *RUNTYPE='MAKESFC'*.

Execute `"dinamica.exe"` por MPI na sua máquina. Em plataformas que possuem sistemas de submissão por lotes (*"batch"*) como o tupa e o Santos Dumont, é necessário executar um script para submeter a tarefa. No tupa, execute o script `./XmitTupa.sh` que exige como único argumento o número de processos MPI (obrigatoriamente 1, no caso *MAKESFC*). Se estiver no Santos Dumont, execute o script `./XmitSDumont.sh` que exige o mesmo argumento.

Execuções bem sucedidas geram arquivo de saída *DIN_<processosMPI>M.out*. Verifique o conteúdo desse arquivo.

A execução no modo *MAKESFC* gera arquivos de dados de superfície na área geográfica e nas datas definidas pelo *RAMSIN* no diretório *runDinamica/data*. Verifique a existência de 16 novos arquivos nesse diretório.

Altere novamente o *RAMSIN*, escolhendo *RUNTYPE='MAKEVFILE'*. Execute novamente a dinâmica com apenas um processo MPI. Este modo de execução gera arquivos com os estados da atmosfera na região geográfica de interesse e nas datas definidas pelo *RAMSIN*. Tais arquivos são gerados a partir das previsões do modelo global e são armazenados no diretório *runDinamica/ivar*. Devem ser gerados cinco pares de arquivos *"iv-brams-XXX"*, um par para cada seis horas de integração. No arquivo de saída (*DIN_1M.out*) avisos sobre a inexistência de arquivos do ar superior e da superfície são esperados e usuais.

Os arquivos gerados por esses dois modos de execução são dados de entrada do modo de previsão (*RUNTYPE='INITIAL'*) e só precisam ser gerados uma única vez para cada *RAMSIN*. Neste ponto do processo, a dinâmica está pronta para ser executada.

Para executar a dinâmica em modo de previsão, edite novamente o arquivo *RAMSIN* escolhendo *RUNTYPE='INITIAL'*. É nesse modo que o paralelismo MPI foi implementado. Escolha o número de processos que considera adequado. Por exemplo, no tupa, execuções com 10 processos MPI no *RAMSIN_60x60* demandam da ordem de 50 segundos.

Os dois arquivos *RAMSIN* fornecidos geram análises a cada seis horas no diretório *ANL*, arquivos pós-processados preparados para visualização pelo *GRADS* nas mesmas datas e horários no diretório *POS* e arquivos de reinício (*"checkpoint-restart"*) com o estado da computação a cada 24 horas de previsão no diretório *HIS*.

Boa sorte em seu trabalho.