UNIVERSIDADE DE PASSO FUNDO
Programa de Pós-Graduação em Computação Aplicada

Dissertação de Mestrado

INTEGRAÇÃO DE GPGPU AO MODELO ETA SEM COMMON BLOCKS

ALEX LIMA DE MELLO

UPF

UNIVERSIDADE DE PASSO FUNDO
INSTITUTO DE CIÊNCIAS EXATAS E GEOCIÊNCIAS
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA

INTEGRAÇÃO DE GPGPU AO MODELO ETA SEM COMMON BLOCKS

Alex Lima De Mello

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Computação Aplicada na Universidade de Passo Fundo.

Orientador: Prof. Marcelo Trindade Rebonatto
Coorientador: Prof. Carlos Amaral Hölbig

Passo Fundo
2020

CIP – Catalogação na Publicação

M527i Mello, Alex Lima de
Integração de GPGPU ao modelo ETA sem common blocks / Alex Lima de Mello. – 2020.
61 f. : il. ; 30 cm.

Orientador: Prof. Dr. Marcelo Trindade Rebonatto.
Coorientador: Prof. Dr. Carlos Amaral Hölbig.
Dissertação (Mestre em Computação Aplicada) – Universidade de Passo Fundo, 2020.

1. Processamento paralelo (Computadores). 2. Programação paralela (Computação). 3. Previsão numérica do tempo (Meteorologia). I. Rebonatto, Marcelo Trindade, orientador. II. Hölbig, Carlos Amaral, coorientador. III. Título.

CDU: 004.272

Catalogação: Bibliotecário Luís Diego Dias de S. da Silva – CRB 10/2241

UPF

PPGCA
Programa de Pós-Graduação
em Computação Aplicada
Instituto de Ciências Exatas e Geociências | ICEG

ATA DE DEFESA DO TRABALHO DE CONCLUSÃO DE CURSO DO ACADÊMICO

ALEX LIMA DE MELLO

Aos trinta dias do mês de março do ano de dois mil e vinte, às quatorze horas, realizou-se, no prédio D1 sala 01, da Universidade de Passo Fundo (UPF), a sessão pública de defesa do Trabalho de Conclusão de Curso "Integração de GPGPU ao modelo ETA sem common blocks", de autoria do acadêmico **ALEX LIMA DE MELLO** do Curso de Mestrado em Computação Aplicada do Programa de Pós-Graduação em Computação Aplicada – PPGCA. Segundo as informações prestadas pelo Conselho de Pós-Graduação e constantes nos arquivos da Secretaria do PPGCA, o aluno preencheu os requisitos necessários para submeter seu trabalho à avaliação. A banca examinadora foi composta pelos doutores Marcelo Trindade Rebonatto, Carlos Amaral Hölbig, Willingthon Pavan, Jorge Luís Gomes e Edson Luiz Padoin. Concluídos os trabalhos de apresentação e arguição, a banca examinadora considerou o candidato APROVADO. Foi concedido o prazo de até quarenta e cinco (45) dias, conforme Regimento do PPGCA, para o acadêmico apresentar ao Conselho de Pós-Graduação o trabalho em sua redação definitiva, a fim de que sejam feitos os encaminhamentos necessários à emissão do Diploma de Mestre em Computação Aplicada. Para constar, foi lavrada a presente ata, que vai assinada pelos membros da banca examinadora e pela Coordenação do PPGCA.

Prof. Dr. Marcelo Trindade Rebonatto– UPF
Presidente da Banca Examinadora
(Orientador)

Prof. Dr. Carlos Amaral Hölbig– UPF
(Coorientador)

Prof. Dr. Willingthon Pavan – UPF
(Avaliador Interno)

Prof. Dr. Jorge Luís Gomes –INPE
(Avaliador Externo)

Prof. Dr. Edson Luiz Padoin -Unijuí
(Avaliador Externo)

Prof. Dr. Rafael Rieder
Coordenador do PPGCA

INTEGRAÇÃO DE GPGPU AO MODELO ETA SEM COMMON BLOCKS

RESUMO

O Eta é um modelo de previsão do tempo utilizado em diversos países, inclusive no Brasil, pelo CPTEC/INPE. Em alguns casos, a execução de modelos de previsão como o Eta pode demorar horas, ou até mesmo dias. Esta dissertação teve como objetivo principal a implementação de tecnologias de GPGPU ao modelo Eta, com o objetivo de reduzir seu tempo de execução. Analisando o código do modelo, foi escolhida uma sub-rotina para implementar duas novas versões da mesma, utilizando CUDA e OpenACC. Analisando os resultados, pode-se observar que as novas implementações apresentam melhor desempenho em relação à versão original quando executado o modelo com baixo número de tarefas MPI, no entanto, o desempenho das mesmas diminui conforme aumenta-se o número de tarefas MPI utilizadas. A versão com CUDA tem melhor desempenho comparada com a versão utilizando OpenACC, no entanto, o tempo de desenvolvimento em OpenACC é consideravelmente menor.

Palavras-Chave: computação paralela, CUDA, OpenACC, previsão numérica do tempo.

GPGPU INTEGRATION TO THE ETA MODEL WITHOUT COMMON BLOCKS

ABSTRACT

Eta is a weather prediction model used in several countries, including Brazil, by CPTEC/INPE. In some cases, the execution of weather prediction models like Eta can take hours, or even days. This dissertation's main objective was the implementation of GPGPU technologies to the Eta model, in order to reduce its execution time. Analyzing the model code, a subroutine was chosen to implement two new versions of it, using CUDA and OpenACC. Analyzing the results, it can be observed that the new implementations show better performance in comparison to the original version when the model is executed with a low number of MPI tasks, however, their performance worsens as the number of threads increases. The CUDA version has better performance compared to the OpenACC version, however, the development time in OpenACC is considerably less.

Keywords: CUDA, NWP, OpenACC, parallel computing.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

NWP – Numerical Weather Prediction

HPC – High-performance Computing

CPTEC – Centro de Previsão do Tempo e Estudos Climáticos

INPE – Instituto Nacional de Pesquisas Espaciais

GPU – Graphical Processing Unit

GPGPU – General-Purpose Graphical Processing Unit

BAM – Brazilian Global Atmospheric Model

HIBU – Hydrometeorological Institute and Belgrade University

PGI – Portland Group Inc.

ALU – Unidade Lógica e Aritmética

WRF – Weather Research and Forecasting Model

API – Aplication Programming Interface

SIMT – Single Instruction, Multiple Thread

NVVP – Nvidia Visual Profiler

UPF – Universidade de Passo Fundo

MPS – Multi-Process Service

SUMÁRIO

1. INTRODUÇÃO

A meteorologia desempenha um papel vital na sociedade moderna, afetando o cotidiano das pessoas. Por meio da previsão do tempo, uma pessoa pode planejar seu dia e se informar sobre áreas de perigo durante desastres naturais. Como exemplo, pode-se citar a aviação e agricultura, por meio de diferentes serviços. Em voos de longa duração, é necessário ter conhecimento sobre as condições atmosféricas da rota e dos locais de escala, para que seja garantida a segurança da aeronave no ar, ou ainda a visibilidade durante o pouso [1].

A utilização de dados climáticos é especialmente importante para a agricultura, pois o cultivo em larga escala depende diretamente das condições climáticas, como a disponibilidade de água, luz solar e calor. Além disso, da mesma forma que a variação do clima pode diminuir o rendimento da colheita, a falta de conhecimento meteorológico pode levar o agricultor a adotar estratégias mais conservadoras, a fim de reduzir riscos [2].

Modelos de previsão numérica do tempo (*Numerical Weather Prediction* - NWP) são ferramentas que trabalham com uma grande quantidade de dados e possuem custo computacional elevado. Estes modelos são compostos por equações que formam um modelo matemático da atmosfera. Com base em dados atmosféricos coletados, conseguem prever variáveis como temperatura, pressão, precipitação e vento para um determinado período de tempo.

Devido à grande quantidade de dados necessários para prever os avanços atmosféricos, modelos de NWP necessitam uma capacidade de processamento elevada, tornando a execução dos mesmos um processo que, em alguns casos, pode demorar vários dias. Buscando uma redução no tempo de execução, são implementados diferentes recursos de computação de alto desempenho.

Computação de alto desempenho (*High-performance Computing* - HPC) se refere a habilidade de processar dados utilizando capacidade de processamento superior às possíveis em computadores normais. Para tal, são utilizados supercomputadores ou *clusters*. O desenvolvimento de aplicações para HPC normalmente consiste na utilização de programação paralela e/ou distribuída, dependendo do tipo de sistema [3].

Existem duas classificações principais para modelos de NWP: globais e regionais. Modelos de NWP regionais realizam a previsão do tempo para uma área específica, definida pelo usuário, estes modelos normalmente utilizam uma resolução mais detalhada e necessitam de informações de condições de borda, ou seja, informação sobre a área ao redor da definida pelo usuário. Modelos de NWP globais realizam a previsão para todo o planeta, porém com menor resolução [4].

O modelo Eta é um modelo de NWP regional utilizado para propósitos operacionais e de pesquisa em vários países. No Brasil, o modelo Eta é utilizado no Centro de Previsão do Tempo e Estudos Climáticos (CPTEC)/Instituto Nacional de Pesquisas Espaciais (INPE) de forma operacional, abrangendo a América do Sul. De acordo com Chou et Al. [5], o modelo é utilizado pelo INPE desde 1997 para previsões meteorológicas e, desde 2002 para previsões do estado do tempo.

O Eta é utilizado para gerar diversas previsões diariamente, com resoluções, prazos, e áreas diferentes, toda a *suite* operacional do modelo Eta pode ser observada na Tabela 1.

Tabela 1. *Suite* operacional do modelo Eta [6].

Prazo de previsão	Resolução horizontal	Frequência	Abrangência
3 dias	1km	2x/dia	Entre RJ e SP
3 dias	5km	2x/dia	Sul e Sudeste
11 dias	15km	2x/dia	América do Sul e Central
11 dias	40km	1x/dia	América do Sul e Central
60 dias	40km	1x/dia	América do Sul e Central
135 dias	40km	1x/mês	América do Sul e Central

O uso de técnicas de HPC, como programação paralela e programação distribuída, é cada vez mais populare em aplicações científicas, para fazer uso de recursos físicos como supercomputadores e *clusters*, que são capazes de executar diferentes operações simultaneamente. Aplicações são analisadas e divididas em diversas partes, que podem ser processadas de forma concorrente em um computador, ou em diversos computadores conectados por rede, com o objetivo de reduzir o tempo total de processamento.

Unidades de processamento gráfico (*Graphical Processing Units* - GPUs) são dispositivos criados para acelerar o processamento de imagens. Devido à tarefa para qual foram criadas, GPUs possuem uma estrutura massivamente paralela, desenvolvida para processar simultaneamente diferentes regiões de uma imagem. Com a evolução da tecnologia, esse poder de processamento paralelo passou a ser utilizado também para processamento de propósito geral (*General-Purpose Graphical Processing Unit* - GPGPU).

Com origem para processamento de imagens, GPUs possuem uma área de controle reduzida, em função de maior capacidade de processamento lógico e aritmético. Devido a esta limitação, GPUs são utilizadas para operações onde são realizadas as mesmas operações sobre diferentes dados, tais operações são chamadas "*single instruction, multiple data* (SIMD)". Como exemplo pode-se citar operações matemáticas vetoriais, que compõem grande parte do modelo Eta e outros modelos de NWP.

No entanto, existem algumas dificuldades quanto a integração de GPGPU. Como citado anteriormente, modelos de NWP processam uma grande quantidade de dados. A comunicação destes dados entre a GPU e o restante da aplicação é custosa, pois a comunicação entre o computador e a GPU é realizada por um barramento de banda limitada.

Outra dificuldade está na compatibilidade entre novos recursos de computação e *softwares* antigos. Em alguns casos, a existência do *software* sobre o qual se está trabalhando antecede as tecnologias para GPGPU utilizadas. Como por exemplo CUDA, lançado inicialmente em 2007, mais de 20 anos após a escrita do modelo Eta. Um recurso utilizado pelo modelo Eta e incompatível com algumas novas tecnologias de computação paralela é o *Common Block*.

Common blocks são blocos de memória compartilhada que podem ser utilizados por diferentes sub-rotinas, usados originalmente como uma alternativa a variáveis compartilhadas, devido a inexistência do recurso em versões anteriores da linguagem Fortran. Em [7] foram identificadas dificuldades quanto a utilização de CUDA e outros recursos em conjunto com *common blocks*. Devido a isto, este trabalho utiliza uma nova versão do modelo Eta, sem *common blocks*, disponibilizada pelo CPTEC.

Considerando a importância do modelo Eta e sua compatibilidade com processamento massivamente paralelo, este trabalho tem como objetivo modificar a implementação do modelo Eta para realizar parte de seu processamento utilizando GPGPU. Com base na versão experimental do modelo, disponibilizada pelo INPE/CPTEC, na qual foi removido o uso de *common blocks*.Para tal, foi escolhida uma sub-rotina propícia à paralelização com GPU, e foram criadas versões da mesma utilizando as tecnologias CUDA e OpenACC.

A alteração proposta por este trabalho tem como objetivo reduzir o tempo necessário para a execução do modelo e investigar o uso das tecnologias testadas em outras partes do modelo, possibilitando que o resultado da previsão do tempo seja obtido de forma mais ágil e, como consequência, possibilitando um melhor aproveitamento dos recursos computacionais.

Este trabalho é composto por cinco capítulos. No segundo capítulo é apresentada uma revisão sobre modelos NWP, Fortran, compiladores, GPGPU, OpenACC e CUDA. No terceiro capítulo é discutida a metodologia utilizada, a análise do modelo, a forma de paralelismo, a implementação da sub-rotina escolhida com OpenACC e CUDA e as evoluções realizadas sobre a mesma, bem como recursos de otimização estudados. O quarto capítulo contém os testes realizados e uma discussão sobre os mesmos. E finalmente, o capítulo cinco contém as considerações finais do trabalho e possíveis trabalhos futuros.

2. REVISÃO DA LITERATURA

Este capítulo tem por objetivo introduzir tópicos relevantes à compreensão deste trabalho. É feita uma breve introdução sobre modelos de previsão e a linguagem fortran. Em seguida é explicado o conceito de GPGPU e o funcionamento das ferramentas utilizadas, como CUDA e OpenACC.

2.1 MODELOS DE PREVISÃO NUMÉRICA DO TEMPO

Modelos de NWP são formados por conjuntos de equações que descrevem o fluxo de fluidos na atmosfera. Com base em parâmetros e dados iniciais, modelos de NWP realizam a previsão do tempo para um determinado período de tempo no futuro [4].

Existem dois tipos de modelos de NWP: modelos espectrais e modelos em ponto de grade. Ambos são baseados no mesmo conjunto de equações, porém representam e solvem as equações de forma diferente. Enquanto modelos espectrais utilizam funções de onda, modelos de pontos de grade representam dados de forma discreta [8].

Em um modelo em ponto de grade os dados necessários para a execução destes modelos são organizados em grades tridimensionais, onde cada elemento corresponde a uma fração da área a ser processada, sobre esta grade são aplicados métodos numéricos para extrapolar as mudanças na atmosfera.

Os modelos de NWP são, principalmente, divididos entre modelos globais e regionais. Modelos de previsão globais realizam a previsão das condições meteorológicas para todo o planeta, enquanto modelos regionais realizam a previsão apenas para uma área específica. Modelos regionais são integrados para todas as escalas temporais, porém com resolução mais detalhada [4]. Um exemplo de modelo global é o Brazilian Global Atmospheric Model (BAM) [9], desenvolvido pelo CPTEC para previsão do tempo e clima, operacional desde janeiro de 2016.

O Eta é um modelo de NWP regional descendente do modelo Hydrometeorological Institute and Belgrade University (HIBU) da década de 1970, sendo reescrito para utilizar o esquema de advecção horizontal estilo Arakawa de Janjic em 1984 [10], e para utilizar a coordenada vertical eta [11].

As equações do modelo Eta são discretizadas pela grade E de Arakawa e, dando nome ao modelo, pela coordenada vertical eta. A principal vantagem da coordenada vertical Eta é a redução de erros em cálculos obtidos a partir de derivadas horizontais [12], tornando o modelo mais preciso em regiões íngremes, como os Andes.

O modelo tem como condição inicial as informações relativas à área sobre a qual sera realizada a previsão, obtidas por meio de análise estatística, e recebe como parâme-

tros a data inicial e o período de tempo a ser previsto. Devido a evolução da atmosfera, é necessária a atualização das condições de borda, isto acontece a cada seis horas, com os dados da previsão do modelo global [12]. Após a execução do modelo, tem-se o resultado da previsão, composto por variáveis como temperatura do ar, umidade específica e pressão atmosférica.

2.2 FORTRAN

Fortran é uma linguagem de programação utilizada majoritariamente pela comunidade científica. Fortran foi criada como alternativa a linguagem Assembly, para reduzir o tempo de preparo de problemas científicos para o computador IBM 704. A linguagem, inicialmente proposta por John W. Backus em 1957, ganhou popularidade com o surgimento de compiladores capazes de otimizar algoritmos e produzir um desempenho comparável ao de Assembly [13].

2.2.1 Compiladores

Por ser uma linguagem de programação voltada à aplicações científicas, existem diversos compiladores compatíveis com as principais tecnologias de alto desempenho. A lista a seguir mostra alguns dos compiladores mais utilizados compatíveis com OpenMP e MPI [14].

- gfortran (GNU)
- pgfortran (PGI)
- ifort (intel)
- XL Fortran (IBM)
- Cray (Cray)

Dos compiladores citados acima destaca-se o pgfortran, desenvolvido pela Portland Group Inc. (PGI), empresa conhecida por desenvolver e comercializar compiladores de alto desempenho para Fortran, C e C++. Adquirida pela Nvidia em 2013, PGI atualmente desenvolve ferramentas e compiladores sob a marca “PGI Compiler and Tools” [15].

A linguagem Fortran é utilizada na implementação de diversos softwares de cunho científico, destacando-se modelos de simulação computacional. O modelo Eta é escrito em Fortran e deve ser compilado com um compilador PGI compatível.

2.2.2 Common Blocks

Em Fortran77, onde não há variáveis globais, faz-se uso de *common blocks*, que são blocos de memória compartilhada utilizados para passar argumentos entre diferentes sub-rotinas, sem que seja necessário passá-los como parâmetro [16]. Na Figura 1 tem-se a declaração de um *common block* utilizado pelo modelo Eta, onde são declaradas variáveis que serão utilizadas durante a execução do mesmo.

```
1  !------------------------------------------------------------------------
2        COMMON/DYNAM/ &
3        DY ,CPGFV ,EN ,ENT ,R,PT ,F4D ,F4Q ,EF4&
4        ,DETA(LM),RDETA(LM),AETA(LM),F4Q2(LM),ETA(LP1),DFL(LP1) &
5
6        ,EM       (JAM),EMT      (JAM) &
7
8        ,DX       (IDIM1:IDIM2 ,JDIM1:JDIM2),WPDAR    (IDIM1:IDIM2 ,JDIM1:JDIM2) &
9        ,CPGFU (IDIM1:IDIM2 ,JDIM1:JDIM2),CURV       (IDIM1:IDIM2 ,JDIM1:JDIM2) &
10       ,FCP      (IDIM1:IDIM2 ,JDIM1:JDIM2),FDIV       (IDIM1:IDIM2 ,JDIM1:JDIM2) &
11       ,F        (IDIM1:IDIM2 ,JDIM1:JDIM2) &
12       ,DDMPU(IDIM1:IDIM2 ,JDIM1:JDIM2),DDMPV    (IDIM1:IDIM2 ,JDIM1:JDIM2) &
13       ,FAD      (IDIM1:IDIM2 ,JDIM1:JDIM2)
14 !------------------------------------------------------------------------
15
16
```

Figura 1. Exemplo de *common block*.

O *common block* apresentado na Figura 1 pode ser encontrado na sub-rotina VTADV do modelo Eta. *Common blocks* podem ou não possuir nomes, caso não tenha nome, o mesmo deve ser declarado no programa principal, além disso, um programa pode conter apenas um *common block* sem nome.

Na linha dois da Figura 1 tem-se o comando “COMMON”, declarando o *common block*, em seguida, o parâmetro entre “/” define o nome do *common block*, no caso “DYNAM”. Após isso são declaradas as variáveis que compõem o bloco, separadas por vírgula. Lembrando que o símbolo “&” indica quebra de linha, os valores entre parênteses representam os índices iniciais e finais de cada *array*.

Como pode ser visto na Figura 1, o tipo das variáveis não é explicitamente declarado no *common block*, além disso, são declaradas variáveis escalares e *arrays* em um mesmo bloco. A utilização de *common blocks* dificulta a legibilidade do código e causa outros problemas na paralelização do mesmo. Para utilizar um *common block*, usa-se o comando “INCLUDE” seguido pelo nome do arquivo onde o mesmo está armazenado, como pode ser visto na Figura 2.

Na Figura 2 temos a utilização de *common blocks* em um código Fortran. Na linha três pode se observar o nome de arquivo “COMM_DYNAM.f90”, este contém o *common block* “DYNAM”, mostrado na Figura 1.

```
1     INCLUDE " COMM_CTLBLK .f90"
2     INCLUDE " COMM_MASKS .f90"
3     INCLUDE " COMM_DYNAM .f90"
4     INCLUDE " COMM_VRBLS .f90"
5     INCLUDE " COMM_CONTIN .f90"
6     INCLUDE " COMM_PVRBLS .f90"
7     INCLUDE " COMM_CLDWTR .f90"
8     INCLUDE "COMM_INDX.f90"
9     INCLUDE " COMM_NHYDRO .f90"
10
```

Figura 2. Exemplo da utilização de *common blocks* em Fortran90.

Common block é considerado obsoleto em Fortran90, e para que seja possível o avanço de estudos com CUDA, OpenCL, OpenACC e OpenMP é necessário reescrever o código para fazer uso de módulos como alternativa. Tal versão já existe [17], compilando o modelo com a mesma é possível seguir com planos de paralelização do Eta com CUDA, OpenMP e outras soluções similares. O uso de *common blocks* no modelo Eta impossibilita a implementação de algumas soluções, como por exemplo CUDA [18] e OpenMP [7].

2.3 GPGPU

GPUs são dispositivos criados para auxiliar no processamento de imagens em tempo real, GPGPU é a utilização de GPUs para computação de propósito geral, como processamento de dados.

Enquanto a arquitetura da CPU é composta por poucos núcleos, grande quantidade de memória *cache* e uma área de controle maior, GPUs são voltadas a paralelismo massivamente paralelo. A arquitetura da GPU contém menor área de controle e menos memória *cache* disponível, porém, possui maior área dedicada a unidades lógicas e aritméticas (ALU). A GPU é composta por centenas de núcleos, capazes de processar milhares de *threads* simultaneamente [19]. A diferença entre a arquitetura da GPU e CPU pode ser observada na Figura 3.

Figura 3. Diferença de arquitetura entre CPU (à esquerda) e GPU (à direita) [20].

Como se pode observar na Figura 3, A GPU possui maior área dedicada às ALUs em relação à CPU, por ter menor unidade de controle, as *threads* da GPU são limitadas a executar a mesma tarefa.

Aplicações que utilizam mais de um tipo de processador, como por exemplo CPU e GPU, são chamadas aplicações heterogêneas. Para tais aplicações, é utilizado o termo *host* para se referir à CPU e a sua memória, e o termo *device* para se referir ao dispositivo auxiliar e sua memória. No caso de aplicações GPGPU, o *device* se refere à GPU.

O *device* possui sua própria memória, separada da memória do *host*, desta forma, o fluxo de uma aplicação utilizando GPGPU geralmente é composto por três etapas principais: copiar os dados necessários para o *device*; executar o processamento no *device*; copiar os resultados para o *host*.

As tecnologias de GPGPU estudadas para este trabalho foram OpenACC, OpenCL e CUDA. Enquanto CUDA e OpenACC são compatíveis com o compilador atualmente utilizado pelo modelo Eta (pgfortran), OpenCL ainda não é suportado pelo mesmo.

Existem diversos trabalhos que exploram a utilização de GPGPU para aplicações de NWP, tanto de forma parcial como completa.Os modelos de NWP mais utilizados entre os artigos encontrados são o *Weather Research and Forecasting Model* (WRF) e o ASUCA, ambos modelos de mesoescala que utilizam pontos de grade. Alguns dos trabalhos utilizando o WRF são:

- Michalakes e Vachharajani [21]. Microfísica de nuvens em CUDA C,
- Ruetsch et al. [22]. Radiação de ondas longas com CUDA Fortran,
- Mielikainen et al. [23]. Esquema de radiação de ondas curtas de Goddard utilizando CUDA C.

Ainda desenvolvido sobre o WRF, o trabalho de Vanderbauwhede e Takemi [24], investiga a viabilidade da implementação utilizando GPGPU no modelo WRF de forma parcial. No trabalho é utilizado OpenCL para a paralelização de trechos de código passíveis de aceleração utilizando a GPU. Neste trabalho foi identificado a largura de banda de memória para comunicação entre CPU e GPU como fator limitante do *speedup* obtido.

Dos trabalhos desenvolvidos com base no modelo ASUCA, destaca-se o desenvolvido por Müller e Aoki [25], onde é desenvolvida uma solução chamada *Hybrid Fortran*, uma extensão da linguagem capaz de gerar código para execução GPGPU sem que seja necessário a reescrita do modelo. De acordo com o artigo, esta solução gera um aumento de aproximadamente 5% no tamanho do código.

2.3.1 OpenACC

OpenACC [26] é uma API desenvolvida pela Cray e Nvidia com objetivo de simplificar a programação de algoritmos em sistemas heterogêneos, porém diferente de CUDA, não possibilita a manipulação direta das operações realizadas pelo *kernel*.

OpenACC faz uso de diretivas de compilador, de forma similar a OpenMP: pelo uso de diretivas de compilador disponibilizadas pela API, o programador indica Laços de repetição e outros blocos de código a serem executados. Diferentemente de OpenMP, OpenACC pode ser usado para ambos CPUs e GPUs [27], definido pela *flag* "-ta" (*target*) durante a compilação. A seguir são apresentadas algumas das *flags* disponíveis.

- "-ta=*multicore*", gera código paralelo para CPUs multicore;
- "-ta=nvidia", para GPUs da Nvidia;
- "-ta=radeon", para GPUs Radeon;
- "-ta=*host*", gera código sequencial para as seções paralelas.

Além destas, pode-se combinar *flags*, compilando o código com o comando "-ta=nvidia,host", o executável criado utilizaria GPU da Nvidia para executar o programa e, caso não encontrada, executaria de forma sequencial [28].

Para utilizar OpenACC basta utilizar um compilador C, C++ ou Fortran compatível com a API, inserindo as diretivas de paralelização nos trechos de códigos desejados, como apresentado na Figura 4.

```
1       !$acc  parallel    loop  reduction (max:error)
2       do j=1,m-2
3          do i=1,n-2
4             Anew(i,j) =    0.25 _fp_kind   * ( A(i+1,j       ) + A(i-1,j       ) + &
5                                                 A(i    ,j -1)   + A(i    ,j+1)   )
6             error   = max( error ,   abs(Anew(i,j)-A(i,j)) )
7          end do
8       end do
```

Figura 4. Exemplo de paralelização com OpenACC em Fortran90.

A Figura 4 contém um trecho de código onde um laço de iteração é paralelizado com OpenACC. Na linha um, a diretiva "!$acc parallel loop" indica ao compilador que o laço deverá ser paralelizado, enquanto "reduction (max:error)" indica uma operação a ser realizada sobre a variável "error", no caso, será armazenado na variável o maior valor encontrado para a mesma durante a execução da seção paralela. As transferências de memória ocorrem de forma automática quando necessário, salvo quando explicitadas pelo programador por meio de diretivas. Observe também que o código é paralelizado da mesma forma independentemente da *target* utilizada.

2.3.2 CUDA

CUDA é uma API criada pela Nvidia para programação paralela com o uso de GPUs. CUDA oferece ao programador controle sobre copias de memória e o código executado na GPU, sendo possível ajustar o código de forma específica para dada situação. Desta forma, quando comparado com OpenACC, é possível que CUDA apresente desempenho superior em alguns casos [29].

O código a ser executado na GPU (*device*) é denominado *kernel*. Antes de executar o *kernel*, é necessário que os dados a serem utilizado pelo mesmo estejam presentes na memória do device, para isso, é necessário copiar os dados da CPU (*host*) para o *device*, após a execução do mesmo, copia-se o resultado do *device* para o *host*.

Em CUDA, cada *kernel* é executado por um *grid*. O *grid* é uma estrutura formada por um ou mais blocos de *threads*, organizados em uma, duas ou três dimensões, como pode ser observado na Figura 5. A quantidade de *threads* em um bloco e a quantidade de blocos que compõem o *grid* são definidos pelo programador, de acordo com os limites da arquitetura.

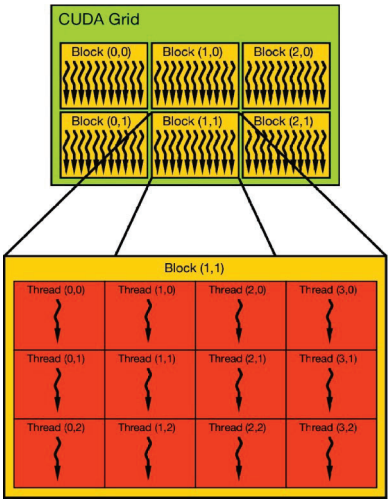

Figura 5. Exemplo de um *grid* bidimensional. A quantidade de *threads* por bloco e a quantidade de blocos no *grid* são definidos pelo programador [30].

De acordo com a Figura 5, o *grid* (verde), é composto por seis blocos de *threads* (amarelo), organizados em duas dimensões. Cada bloco contém 12 *threads* (laranja), neste caso também dispostas em duas dimensões, porém o número de dimensões do bloco e do *grid* não necessita ser o mesmo. Por exemplo: os blocos no *grid* podem ser organizados em duas dimensões, e as *threads* no bloco em uma dimensão.

Cada *thread* CUDA é identificada pela sua posição nas dimensões x, y e z, definida por sua localização no bloco. Dessa forma, tem-se um processamento *Single Instruction, Multiple Thread* (SIMT) [31], isto é, toda *thread* executa a mesma sequência de comandos, porém com valores diferentes.

Para identificar o índice das *threads* no bloco, o índice dos blocos no *grid*, e o tamanho do *grid*, são utilizadas “variáveis *built-in*” do CUDA, que são:

- threadIdx%x, threadIdx%y e threadIdx%z, identificam o índice da thread no bloco a que pertence, para as dimensões x, y e z, respectivamente;
- blockIdx%x, blockIdx%y e blockIdx%z, identificam o índice do bloco no *grid* para as dimensões x, y e z, respectivamente;
- blockDim%x, blockDim%y e blockDim%x, identificam o tamanho do bloco nas dimensões x, y e z, respectivamente, ou seja, a quantidade de *threads* em cada dimensão do bloco.

Na Figura 6 tem-se um exemplo de *kernel* onde é realizada a multiplicação de valores entre diferentes matrizes, armazenando o resultado.

```
1   attributes (global)  subroutine  hyper(d_a ,d_b , d_c , d_d ,  MYIS ,  MYIE ,  MYJS2 ,
    MYJE2)
2   implicit  none
3   integer ,  value  ::  MYIS ,MYIE ,MYJS2 ,MYJE2
4   real ,device  ::  d_a (: ,:), d_b (: ,:,:), d_c (:,: ,:), d_d (:)
5   integer  :: i,j
6
7   i = (( blockIdx%x  - 1)  *  blockDim%x) + threadIdx %x
8   j = (( blockIdx%y  - 1)  *  blockDim%y) + threadIdx %y
9   if(i<1  .OR. i>(MYIE -MYIS  +1) .OR. j<1  .OR. J>(MYJE2 -MYJS2 +1))  return
10  d_a(i,j) =  d_b(i,j ,1)  *  d_c(i,j ,1)  *  d_d (1)
11  end subroutine  hyper
12
```

Figura 6. Exemplo de *kernel* CUDA.

O *kernel* mostrado na Figura 6 multiplica índices das matrizes e armazena em d_a, para isto, é utilizado o índice de cada *thread* nas dimensões I e J em conjunto com o índice e o tamanho do bloco, para calcular o índices para as matrizes, como descrito nas linhas sete e oito.

Em seguida, na linha nove as *threads* com índices fora do intervalo computado pelas matrizes são finalizadas. Todas as *threads* executam a operação descrita na linha 10 de forma paralela, e então o *kernel* é finalizado (linha 11).

3. METODOLOGIA

Neste capítulo são descritos a análise realizada sobre o código do modelo Eta e o modelo de implementação proposto à ser implementado, com objetivo de obter um ganho de desempenho.

3.1 ANÁLISE DO CÓDIGO DO MODELO ETA

Em estudos anteriores realizados pelo autor [32] e Henrique Flores [7] foram realizadas análises do código do modelo Eta, buscando pontos de melhorias no mesmo, identificando partes que necessitam maior tempo de processamento e que possam ser implementadas utilizando recursos de GPGPU. Foi identificada a sub-routina VTADV como sendo a mais computacionalmente intensiva, correspondendo a aproximadamente 16% do tempo de execução do modelo.

A sub-rotina VTADV é composta por diversos laços de iteração, que aparentam ser ideais para processamento em GPU. Para tal, foram estudadas as tecnologias OpenACC e CUDA para implementar uma versão heterogênea, do modelo, onde o processamento do VTADV é realizado na GPU, buscando um possível ganho de desempenho sem alterar os algoritmos utilizados pelo modelo.

A nova versão do modelo Eta disponibilizada é de caráter experimental, foram disponibilizados apenas os códigos-fonte da nova versão, sendo necessário adaptar os *scripts* e *makefiles* utilizados pela versão operacional do modelo. Devido a isto, algumas funcionalidades do modelo não foram restauradas, como por exemplo, a capacidade de alterar o tamanho da área de previsão.

Em alguns casos, a versão do modelo Eta disponibilizada apresenta algumas diferenças em relação à versão operacional do modelo (utilizando *common blocks*), estas diferenças entre as duas versões também foram identificadas pelo CPTEC [33], e não interferem na implementação realizada neste trabalho.

3.2 PARALELISMO DE DOIS NÍVEIS UTILIZANDO GPGPU

Considerando a versão operacional do modelo Eta como paralelismo de um nível, o modelo proposto por este trabalho consiste em implementar um segundo nível de paralelismo. Desta forma o processamento do modelo é primeiramente dividido em tarefas MPI, e parte do processamento de cada tarefa é realizado em GPU, como pode ser observado na Figura 7.

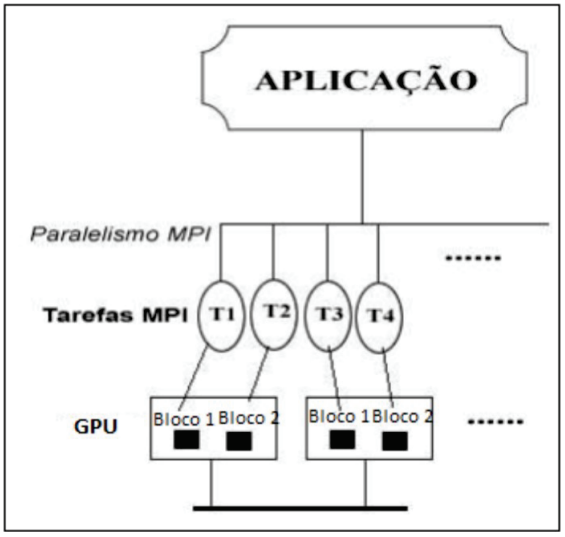

Figura 7. Modelo de solução proposto com dois níveis de paralelismo, o primeiro MPI, e o segundo utilizando OpenACC ou CUDA para processamento em GPU [7].

Como exemplificado na Figura 7, de acordo com este modelo, o processamento do Eta é primeiramente dividido entre tarefas MPI, quando o modelo atinge o ponto determinado, o processamento daquela parte do modelo é realizado pela GPU, utilizando a tecnologia escolhida.

Tanto o modelo de solução proposto utilizando OpenACC quanto CUDA funcionam de maneira similar, a diferença entre ambos se encontra nas qualidades de cada tecnologia: a implementação com CUDA requer maior controle e conhecimento sobre o código fonte, porém em certos casos pode apresentar melhor desempenho.

Para ambas as implementações realizadas, os dados iniciais são copiados para a memória do *device* no início da execução da sub-rotina e no final da mesma, são copiados os resultados para o *host*. Tem-se como objetivo minimizar as cópias de memória, pois são as responsáveis pelo maior custo de desempenho de aplicações em GPU. Os dados são mantidos na memória do *device* quando possível, e atualizados quando necessário.

3.3 IMPLEMENTAÇÃO DA VERSÃO DA SUB-ROTINA VTADV COM CUDA

A utilização de CUDA para paralelização de aplicações comumente se da pela utilização de *threads* CUDA para executar simultaneamente o que seria um laço de iteração. Para tal, é calculada a quantidade de *threads* necessárias.

O índice da iteração correspondente a cada *thread* é calculado com base no índice da *thread* no bloco, o tamanho de cada bloco, e o índice do bloco no *grid*. Assim é possível obter um índice contínuo independente do bloco em que a *thread* se encontra, possibilitando a utilização para computar valores com matrizes maiores que o máximo de *threads* por

bloco. As *threads* com índice calculado fora do intervalo necessário são finalizadas antes de iniciar o processamento da matriz.

O código da sub-rotina foi alterado para execução em GPU utilizando CUDA. A nova versão contém 44 *kernels* cuda e em torno de 2000 linhas, aproximadamente duas vezes o tamanho do original. Após termos uma versão funcional com processamento em GPU, foram estudadas diversas formas para reduzir o tempo de execução da mesma.

Buscando otimizações na nova versão desenvolvida, foi feita a análise da mesma com o uso do *profiler* "NvProf", os resultados do *Nvprof* foram importados e analisados utilizando a ferramenta "*Nvidia Visual Profiler* ".

O *Nvidia Visual Profiler* -NVVP é uma ferramenta utilizada para analisar o código do programa executado e oferecer ao desenvolvedor sugestões de como melhorar seu desempenho por meio de uma interface visual [34].

Devido a quantidade de *scripts*, códigos-fonte e outros recursos utilizados pelo Eta, foi julgado mais simples adaptar o *script* de execução do modelo para utilizar o *profiler* por linha de comando, utilizando o NVVP apenas para analisar os arquivos gerados pelo Nvprof.

Como resultado, foram identificados dois gargalos responsáveis por limitar significativamente o desempenho da nova versão: o tempo gasto com cópias de dados e a baixa utilização da GPU.

Para realizar a execução do Eta com o profiler foi adaptado o script ".../worketa/eta/scripts/run.com_real_mpi". O *profiler* pode ser utilizado para coletar várias métricas dependendo dos parâmetros utilizados para sua execução, primeiramente foi executada uma análise superficial, que possibilita visualizar a linha do tempo da aplicação e o tipo de operações que estão sendo realizadas em cada momento, como por exemplo processamento do kernel, cópia de memória, chamadas à API, etc. Para isto, foi utilizado o comando descrito na Figura 8.

```
1
2
3
```

```
mpirun -n ${npr}  nvprof  -o / etaProject/PROF/eta .%p.nvvp  \
${ Model_exe }> ${Eta_run }/ saida_Eta.out
```

Figura 8. Comando de execução do modelo Eta utilizando o *profiler* nvprof para coletar dados sobre a utilização da GPU.

Na Figura 8 é demonstrada a utilização do *profiler* em conjunto com a execução do modelo Eta. O comando "nvprof" indica a utilização do *profiler*, a *flag* "-o" seguida por um endereço indica onde será gravado o arquivo com o resultado, o nome do arquivo de resultado será "eta.%p.nvvp", onde "%p" representa o número identificador do processo sobre o qual o *profiler* foi executado. Para que o NVVP possa recomendar otimizações ao código, é necessario utilizar a *flag* "–analysis-metrics", porém sua adição tem um grande

impacto no tempo de execução do modelo, uma execução que demora alguns minutos pode demorar várias horas, dependendo da quantidade de processos.

Utilizando o NVVP foi identificado que ao utilizar declaração implícita do tamanho das matrizes no *kernel* cria chamadas extras à função de cópia de memória. A solução neste caso foi declarar explicitamente o o tamanho de cada dimensão das matrizes.

3.3.1 Reduzindo o impacto de cópias de dados

Com o objetivo de reduzir o tempo gasto com cópias de dados entre a memória do *host* e do *device* foram identificadas matrizes que são utilizadas em outras sub-rotinas do modelo. As matrizes identificadas podem ser categorizadas de três formas: matrizes exclusivas ao VTADV, matrizes que são alteradas apenas no VTADV e matrizes que são alteradas em outras sub-rotinas.

Matrizes que são utilizadas somente no VTADV podem ser mantidas na memória da GPU, desta forma não sendo necessário realizar cópias para cada iteração modelo onde a sub-rotina é invocada. Para tal caso, é realizada uma cópia na primeira execução do VTADV, a matriz é mantida na memória do *device* até o ultimo *timestep* do modelo.

Matrizes que são alteradas apenas no VTADV não necessitam de uma nova cópia no início da sub-rotina, porém, ao final de cada iteração da mesma é necessário copiar o valor computado pelo *device* para o *host*, para que as outras sub-rotinas tenham acesso aos novos valores computados.

Matrizes alteradas em outras sub-rotinas são copiadas para o *device* no início de cada iteração do VTADV. Caso as mesmas sejam alteradas novamente durante a execução do VTADV, é necessário copiar os dados para o *host*no final da iteração, caso contrário, a cópia não é necessária.

Como padrão, as variáveis da GPU se tornam indefinidas ao final de cada iteração da sub-rotina no modelo. Para que isso não aconteça é utilizado o atributo “SAVE” na declaração das mesmas, indicando que os valores serão preservados entre invocações sucessivas do VTADV.

Após esta redução inicial, a nova versão do VTADV apresenta tempo de execução inferior à versão original para certos casos, porém o tempo gasto com cópias de memória entre *host* é *device* ainda é significativo. Na Figura 9 pode ser visualizada a diferença no tempo gasto com cópias de memória entre as duas versões.

A Figura 9 foi retirada da linha do tempo de uma execução do modelo Eta, visualizada pelo NVVP. Na parte esquerda da figura são apresentadas diferentes informações fornecidas pela aplicação, enquanto a parte direita corresponde à *timeline* da aplicação.

Na primeira linha da Figura 9 é descrito o *device* no qual todas as operações seguintes foram executadas, no caso, é uma GPU Tesla K40c, da Nvidia, o número zero

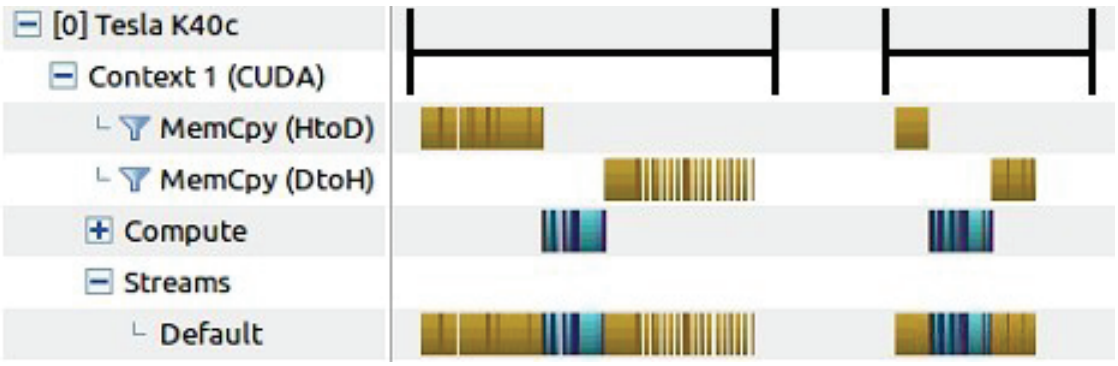

Figura 9. Diferença entre as cópias de memória da versão inicial do VTADV em CUDA (à esquerda) e a nova versão (à direita).

entre colchetes indica que esta é a GPU de índice zero no computador. Na terceira e quarta linha são representadas as cópias de memória entre *host* e *device*, a terceira linha representa as cópias *host*-to-*device*, enquanto a quarta linha representa cópias *device*-to-*host*.

A quinta linha da Figura 9 com título "*Compute*", representa todos os *kernels* executados, caso expandida, é possível observar cada *kernel* em uma linha diferente, com a porcentagem do tempo de processamento correspondente a cada um. A sexta linha expande para mostrar a lista de *streams* utilizadas na execução, no caso temos apenas a *stream Default*, representada na sétima linha.

Na sétima linha da Figura 9 é descrito tudo que foi executado em uma determinada *stream*, como se pode ver na imagem, é mostrado as cópias e kernels na ordem em que as operações foram executadas. Ainda que a quantidade de cópias tenha sido reduzida, o tempo de processamento de *kernels* ainda corresponde apenas a aproximadamente 50% do tempo total de execução do VTADV.

A *timeline* apresentada na Figura 9 mostra duas iterações do VTADV, a primeira iteração representa a versão inicial da implementação de CUDA no VTADV, enquanto a segunda representa a nova versão. Repare que o tempo gasto com cópias é reduzido significativamente, consequentemente, reduzindo o tempo total de execução da sub-rotina.

Para tentar reduzir ainda mais o tempo com cópias de dados, foi implementada uma versão utilizando cópias assíncronas. Cópia assíncrona é um recurso de CUDA onde as cópias são executadas de maneira não bloqueante, ou seja, é possível realizar cópias entre *host* e *device* simultaneamente à execução de um *kernel*.

Para utilizar cópias assíncronas, é necessário que as variáveis do *host* tenham o atributo "*PINNED*", significando que estão em memória não paginada (*page-locked*). Memória *PINNED* possui velocidades de cópia superior em relação a memória paginada, porém seu uso é limitado à quantidade de memória RAM disponível [35, p. 45].

Muitas das variáveis utilizadas pelo VTADV se encontram em módulos também utilizados por outras sub-rotinas. Para utilizar memória "*PINNED*" seria necessário reescrever

os módulos do modelo Eta para utilizarem tal tipo de memória em todo o modelo, alternativa considerada inviável.

Outra possível solução seria criar no VTADV cópias "*PINNED*" das variáveis utilizadas pelo modelo, porém esta solução introduz uma nova camada de cópias de memória entre o VTADV e as demais sub-rotinas do modelo.

3.3.2 Melhorando o desempenho do processamento em GPU

Utilizando o *profiler*, foram identificados os *kernels* com maior tempo de processamento, destes, alguns foram reescritos para conseguir um desempenho superior. No entanto, outros *kernels* apontados pelo *profiles* não puderam ser melhorados devido a dependências, em diversos casos matrizes de três dimensões puderam ser paralelizadas em apenas duas delas, a terceira permanecendo um laço de iteração.

Também foi estudado o uso de *CUDA streams* de diferentes formas. O primeiro teste utilizando cópias assíncronas não obteve resultados positivos. Outra estratégia foi utilizar *streams* para executar *kernels* de uma mesma tarefa MPI concorrentemente, porém não apresentou ganho significativo de desempenho, devido à dependência de dados entre os mesmos.

Foi também implementado o uso de *streams* para diferentes tarefas MPI, desta forma eliminando a dependência de dados, porém tal implementação não funcionou da maneira prevista.

Pesquisando, foram encontradas situações similares, como por exemplo em C, o uso de "*pthreads*" utiliza a mesma *stream* por padrão, fazendo com que *kernels* em diferentes *threads* executem de forma sequencial. A solução para tal caso em C é a utilização da *flag* "–default-stream per-thread" [36].

Fortran não possui uma alternativa equivalente a *flag* utilizada em C, porém a chamada à API "CudaSetStreamDefault" pode ser utilizada para alterar a stream padrão de um processo.

Mesmo utilizando a chamada descrita, a execução com MPI ainda ocorre de forma sequencial, levando a conclusão de que diferentes tarefas MPI não possuem acesso concorrente à GPU, resultado corroborado por [37] e [24].

3.4 IMPLEMENTAÇÃO DA VERSÃO DA SUB-ROTINA VTADV COM OPENACC

A implementação da sub-rotina VTADV com OpenACC é relativamente simples quando comparada com a versão com CUDA, dado que não é necessário reescrever o código da sub-rotina, bastando usar as diretivas disponibilizadas pela API para definir a forma como o código será paralelizado. Como descrito na seção 2.3.1, OpenACC pode

ser utilizado tanto para paralelismo em GPU quanto em CPU, porém devido ao foco deste trabalho, esta implementação utiliza a API apenas para paralelismo em GPU.

Para paralelizar os laços de iteração da sub-rotina foi utilizado a diretiva "*acc parallel loop*," em conjunto com a clausula "*collapse*", conforme o possível, para paralelizar laços aninhados.

Inicialmente foi criada uma versão da sub-rotina utilizando controle automático de memória, no entanto, o tempo desta versão foi consideravelmente maior que o da versão original. Para tentar melhorar o desempenho da versão com OpenACC, foi então estudado uma maneira de reduzir as cópias de memória realizadas, controlando-as de forma explicita.

Para tal, foi utilizado diretivas não estruturadas de dados, pelas quais é possível manter dados na GPU sem a necessidade de criar um escopo. Assim, é possível copiar os dados para a GPU e mantê-los na mesma até o final da execução do modelo.

A estrutura das cópias de dados entre a memória do *host* e do *device* é semelhante à versão com CUDA. Os dados são copiados para o *device* na primeira iteração do modelo. Para as demais iterações, são usadas diretivas para atualizar os dados no *device* e copiar para o *host* as variáveis utilizadas por outras sub-rotinas. As diretivas utilizadas para controlar as cópias de memória foram as seguintes:

1. "*acc enter data copyin*", para alocar e copiar variáveis para o *device*;
2. "*acc update device*", copia uma variável do *host* para o *device*;
3. "*acc update self* ", copia uma variável do *device* para o *host*;
4. "acc exit data copyout", para desalocar variáveis no *device* e copiar variáveis para o *host*.

Para indicar que as variáveis necessárias para a execução de um laço já estão no device, foi utilizada a cláusula "default(present)" como parte da diretiva usada para iniciar a seção paralela.

A utilização de diretivas para controle de memória reduziu significativamente o tempo de execução desta versão, porém ainda assim o tempo obtido é consideravelmente maior em relação ao tempo da versão com CUDA.

4. TESTES E RESULTADOS

Neste capítulo são descritos os testes realizados, o ambiente nos quais os mesmos foram executados, e os resultados obtidos. A lista completa com todos os tempos obtidos durante os testes está disponível nos apêndices, juntamente com as análises estatísticas realizadas.

4.1 AMBIENTE DE EXECUÇÃO

Para a execução dos experimentos apresentados nesta seção foram utilizados recursos disponibilizados pela Universidade de Passo Fundo (UPF), encontrados na sala de realidade virtual do módulo três do UPF Parque.

O ambiente computacional utilizado é composto por três computadores com CPU Intel Core i7-6950X com clock de 3.5GHz, com 10 núcleos físicos e 20 lógicos;32 Gb de memória RAM; Sistema operacional Linux, distribuição Ubuntu versão 18.04 LTS 64 bits. Dois destes computadores equipados com GPU GeForce TITAN X da Nvidia, com 3072 CUDA cores com 12 Gb de memória de interface 384-bits GDDR5 e largura de banda de 336.5 Gb por segundo. O terceiro computador possui uma GPU GeForce Tesla K40 da Nvidia, com 2880 CUDA cores, exclusivos para processamento de dados, 12 Gb de memória, interface 384-bits, com largura de banda de 288 Gb por segundo e outra GPU GTX 660 da Nvidia para controle de vídeo, sem uso no processamento.

Os testes finais foram executados utilizando apenas um computador, equipado com GPU Nvidia Tesla K40, com o objetivo de acelerar os testes e verificar de forma mais clara o impacto da implementação com GPGPU.

As GPUs GeForce TITAN X possuem arquitetura Maxwell, enquanto a Tesla K40 possui arquitetura Kepler. No entanto, ambas apresentam os mesmos limites em relação a quantidade de *threads* por bloco, dimensões dos blocos, e *warp size*. Desta forma, o código desenvolvido não necessita ser otimizado para executar em uma GPU específica, mesmo que as GPUs possuam capacidade de processamento diferente. Para ambas as GPUs é possível ter no máximo 1024 *threads* por bloco, enquanto o limite por dimensão é de 1024 *threads* para as dimensões x e y, e 64 *threads* para a dimensão z.

Para a compilação do modelo Eta foi utilizado o compilador PGI versão 19.4 *Community Edition* para processadores de 64 bits. Para a compilação do modelo com CUDA, foi utilizado o driver CUDA versão 418.40.04, disponível no “CUDA *toolkit* 10.1”. Para compilar o modelo para a execução com CUDA, basta incluir a *flag* “-MCuda” ao arquivo “../eta/src/-configure/make.inc”.

4.2 PARÂMETROS DE EXECUÇÃO

Para comparar os tempos obtidos para as diferentes versões da implementação, foram realizados testes com diferentes quantidades de processos. O período de previsão utilizado foi de 12 horas, e a área de previsão de 101x159 pontos sobre o território brasileiro, com resolução de 10km, como pode ser visto na Figura 10.

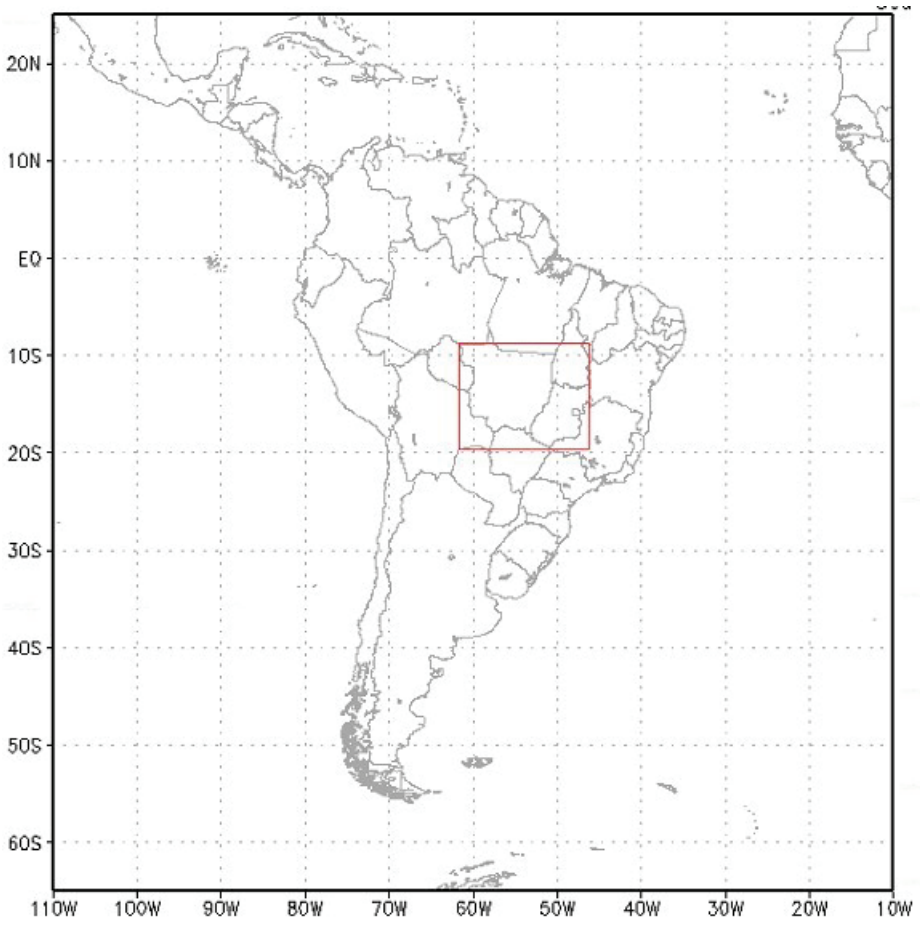

Figura 10. Representação da área de previsão utilizada para a realização dos testes, destacada pelo retângulo vermelho.

A área central, em destaque, da Figura 10 representa a área sobre a qual foi executado o processamento do modelo Eta, a área é expandida a partir de um ponto central, escolhido pelo usuário. Devido a dificuldades em relação a esta versão do modelo, todos os testes foram realizados com apenas uma área de processamento.

A quantidade de processos utilizados é definida pelo uso de duas variáveis que definem entre quantos processos cada eixo será dividido, formando assim, uma matriz de processos, cada um responsável por computar parte da área de previsão, desta forma, a quantidade de processos utilizados se dá pela multiplicação destas duas variáveis, como exemplificado na Figura 11.

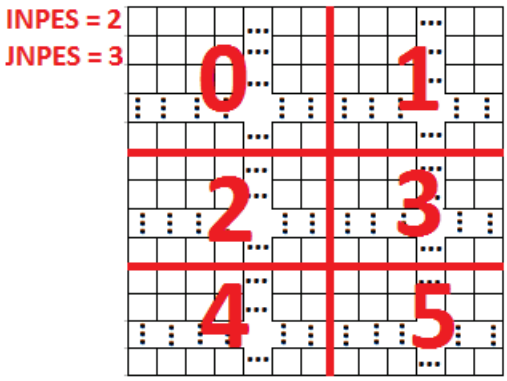

Figura 11. Divisão da área de previsão entre a quantidade de processos, definidos pelas variáveis INPES (horizontal) e JNPES (vertical).

O processo de divisão da área de previsão entre os processos definidos pelas duas variáveis é exemplificado na Figura 11, a área total da imagem é dividia entre dois processos no eixo I, pela variável INPES, e entre 3 processos no eixo J, pela variável JNPES. desta forma o número total de processos utilizados para o processamento da área se dá pela multiplicação das duas váriaveis, no caso, a área do exemplo será dividida entre seis processos. O número de processos utilizado para os testes será exibido no formato de multiplicação, como por exemplo 2x3, indicando que INPES=2 e JNPES=3.

Foram comparados os tempos de execução da sub-rotina VTADV original, com CUDA e com OpenACC, utilizando diferentes números de processos por computador. Para cada teste foram realizadas 40 repetições, e considerado o tempo médio com arredondamento a duas casas decimais.

4.3 RESULTADOS DAS IMPLEMENTAÇÕES COM CUDA

A implementação da sub-rotina VTADV com CUDA compôs a maior parcela de tempo utilizada neste trabalho, logo, será mostrado algumas iterações da mesma, e comparado o tempo de cada em relação a versão original da sub-rotina. Em todas as versões os resultados finais foram comparados e validados com a versão sem a utilização da GPU.

4.3.1 Primeira Versão

A primeira versão da sub-rotina com CUDA executa o processamento das matrizes completamente em GPU. Toda iteração do modelo em que a sub-rotina é executada, todos os dados necessários são copiados para o *device* e, ao final da execução, são copiados novamente para o *host*. Isto possui um alto custo computacional, que é refletido no tempo de execução. Vide Tabela 2.

Tabela 2. Comparação do tempo médio de execução entre a versão original da sub-rotina VTADV e a primeira versão em CUDA.

Processos	Original	Versão 1
1x1	111,97	135,27
1x2	63,22	138,84
1x3	42,55	138,10
2x2	36,45	172,60
2x3	27,04	169,05
3x3	25,32	218,83

Observando a Tabela 2 pode-se observar que o tempo sequencial (1x1) da nova versão é maior que o original. Além disso, o tempo de execução aumenta com a quantidade de processos. Executando com nove processos, o tempo da versão com CUDA é aproximadamente 800% da versão original.

O maior custo de tempo desta versão é o uso desnecessário de transferência de dados. Este tempo pode ser reduzido consideravelmente com a análise das matrizes utilizadas pelo VTADV. Identificando quais variáveis são modificadas ou utilizadas em outras sub-rotinas do modelo, é possível manter certas variáveis na memória da GPU.

4.3.2 Segunda Versão

Nesta versão foi reduzido o número de cópias realizadas entre *host* e *device*. Os dados não alterados em outras sub-rotinas são mantidos na memória da GPU entre iterações do VTADV, dados alterados em outras sub-rotinas são copiados para o *device* no início de cada iteração, e no fim dados utilizados em outras sub-rotinas são copiados para o *host*.

A redução das cópias resulta em um tempo de processamento consideravelmente menor em relação à primeira versão, e em alguns casos, à versão original do VTADV, como mostrado na Tabela 3.

Tabela 3. Comparação do tempo médio de execução entre a versão original da sub-rotina VTADV e a segunda versão em CUDA.

Processos	Versão Original	Segunda Versão
1x1	111,97	30,12
1x2	63,22	28,51
1x3	42,55	24,65
2x2	36,45	28,37
2x3	27,04	34,60
3x3	25,32	44,43

Na Tabela 3 pode ser observado que a versão do VTADV com CUDA apresenta desempenho superior em relação a versão original para até quatro processos. No entanto, o tempo de processamento da versão original é reduzido consideravelmente conforme o

aumento do número de processos, enquanto a versão com CUDA mantém tempo de execução relativamente constante para até quatro processos, tendendo a aumentar para maior número de processos.

Nesta versão é possível observar dificuldade na escalabilidade da implementação em CUDA: os resultados obtidos indicam que diferentes processos estão competindo pelo uso concorrente da GPU, o que causa a execução de *kernels* de maneira sequencial. Consequentemente, quanto maior o número de processos, maior a competição pela utilização da GPU.

4.3.3 VTADV com Cópias Assíncronas de Memória

A utilização de Cópias assíncronas foi estudada com o objetivo de reduzir o tempo de execução da sub-rotina VTADV, porém tal implementação não apresentou os resultados esperados.

Para a implementação de cópias assíncronas foi necessário criar variáveis locais em memória paginada, sendo necessário primeiro copiar os dados das variáveis globais do no modelo Eta, contidas nos módulos, para então efetuar as cópias *host-to-device* de forma assíncrona, o que resulta em uma redução do potencial ganho em desempenho.

Uma solução para tal situação é modificar os módulos do modelo para utilizar memória paginada, no entanto tal solução, além necessitar a alteração de maior parte do modelo Eta, limita o mesmo à quantidade de memória física disponível. Não obstante, a principal limitação da implementação de CUDA ao modelo Eta é a utilização concorrente da GPU por diferentes tarefas MPI. Nesta implementação não foi utilizada execução paralela de *kernels*. Os resultados obtidos podem ser visualizados na Tabela 4.

Tabela 4. Comparação do tempo médio de execução entre as versões original, da segunda versão e a versão utilizando cópias assíncronas de memória.

Processos	Versão Original	Segunda Versão	Cópias Assíncronas
1x1	111,97	30,12	31,12
1x2	63,22	28,51	31,12
1x3	42,55	24,65	27,54
2x2	36,45	28,37	32,68
2x3	27,04	34,60	39,58
3x3	25,32	44,43	39,03

Como pode ser observado na Tabela 4, os tempos obtidos por esta implementação são maiores em relação a Segunda versão para um, dois, três, quatro e seis processos. Para nove processos, a versão utilizando cópias assíncronas apresenta uma redução no tempo de execução em relação a segunda versão com CUDA, no entanto, o tempo obtido é superior à versão original da sub-rotina. Logo, os resultados indicam que a segunda versão

é superior à versão utilizando cópias assíncronas, pois para os casos onde o desempenho obtido com CUDA é superior ao original, a segunda versão apresenta melhor tempo de execução. O desempenho desta versão é prejudicado pela necessidade de criar variáveis “PINNED” para servir de intermediárias entre as variáveis do modelo Eta, contidas em módulos, e as variáveis utilizadas pelo *device*.

4.3.4 VTADV com *Streams*

As Dificuldades em relação ao uso de *streams* foram previamente descritas na seção 3.3.2. O objetivo desta versão foi solucionar o problema de execução concorrente de *kernels* identificado na segunda versão.

Foram estudadas diversas maneiras de implementar o uso de *streams* com o objetivo de forçar a utilização concorrente da GPU entre processos MPI, porém nenhuma foi bem-sucedida. Assim, Esta implementação utiliza *streams* para processamento de *kernels* em uma mesma tarefa MPI. Os resultados podem ser observados na Tabela 5.

Tabela 5. Comparação do tempo médio de execução entre as versões original, a segunda versão e a versão utilizando *streams*.

processos	Versão Original	Segunda Versão	Versão com Streams
1x1	111,97	30,12	30,22
1x2	63,22	28,51	28,54
1x3	42,55	24,65	23,76
2x2	36,45	28,37	28,78
2x3	27,04	34,60	34,93
3x3	25,32	44,43	44,30

A implementação da sub-rotina VTADV com *streams* para execução concorrentes de *kernels* em uma mesma tarefa MPI apresentou desempenho similar ao da segunda versão. Realizando o teste t entre ambas as versões, é obtido um valor-p inferior a 0,05 apenas para a execução com três processos (valor-p = 0,029), indicando que para os outros casos não há uma diferença significativa entre as versões.

Devido a utilização das mesmas matrizes, a maioria dos *kernels* devem ser executados de forma sequencial para garantir a integridade dos resultados. O paralelismo entre os mesmos não pôde ser explorado suficientemente para justificar o uso de *streams*. Desta modo, os tempos obtidos são semelhantes aos da segunda versão.

4.3.5 Considerações sobre os Resultados com CUDA

Como a versão utilizando cópias assíncronas apresentou um aumento no tempo de processamento para a maioria dos casos, e a versão utilizando *streams* não apresentou

diferença significativa na maioria dos casos, a segunda versão é considerada a versão final da implementação do VTADV com CUDA. Os resultados obtidos podem ser observados na Figura 12.

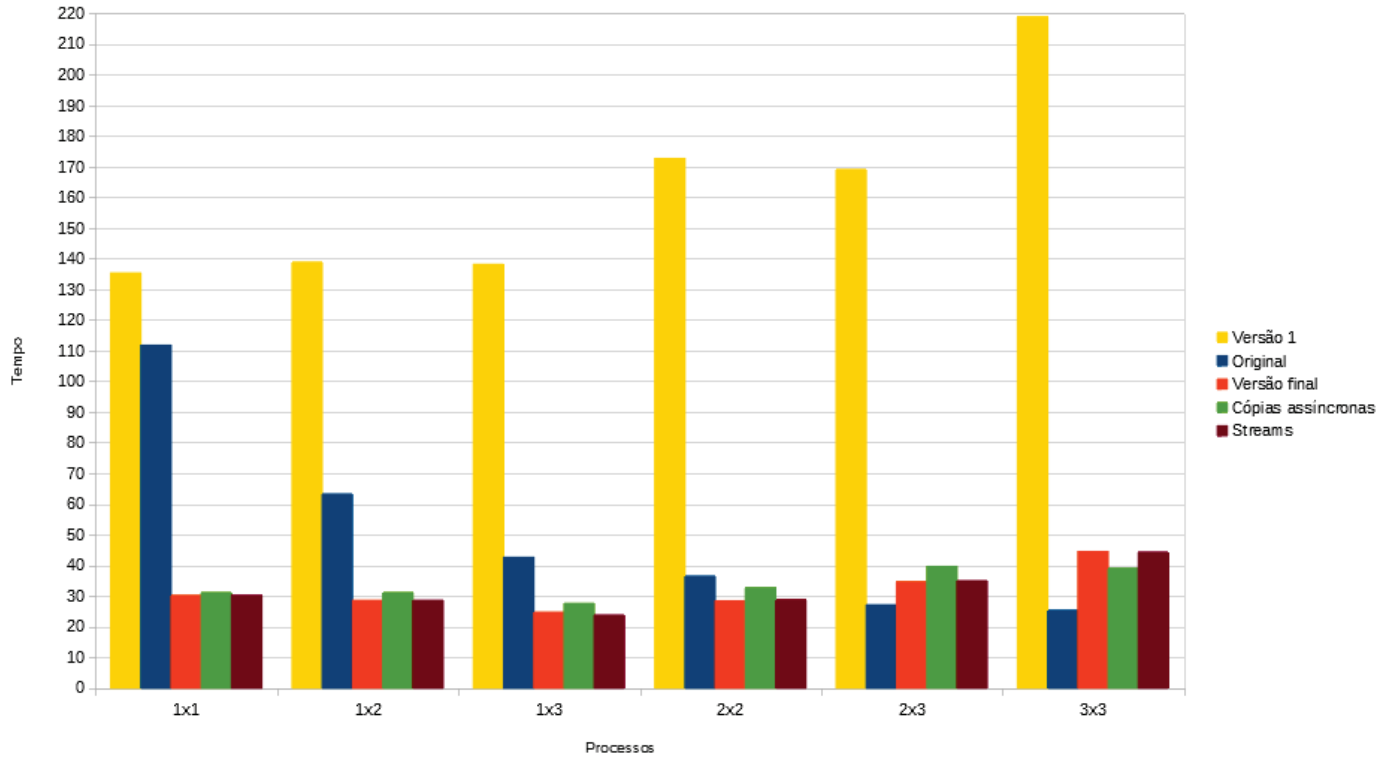

Figura 12. Visualização dos tempos médios de execução da sub-rotina VTADV para as diferentes versões implementadas.

Observando a Figura 12, nota-se a primeira versão implementada com CUDA apresenta tempo de execução drasticamente superior às outras versões, reiterando o impacto do tempo gasto com cópias de memória entre *host* e *device*, suportado pelas conclusões de [24]. As versões seguintes apresentam tempos similares, no entanto, a versão final (4.3.2) apresenta tempo de execução ligeiramente menor em relação as implementações utilizando cópias assíncronas, exceto para os testes utilizando nove processos. Como explicado anteriormente, a diferença entre a versão final e a versão com *streams* é significativa apenas para os testes com três processos.

Ainda analisando a Figura 12, pode-se observar que o tempo de execução da versão original diminui significativamente com o aumento no número de processos. No entanto, as versões implementadas com CUDA apresentam tempos relativamente constante, para até quatro processos o tempo de execução diminui levemente, onde passa a aumentar conforme a quantidade de processos.

Tal situação explicita a competição de tarefas MPI pela utilização da GPU: enquanto na versão original diferentes processos utilizam a CPU de forma concorrente, nas versões com CUDA, diferentes processos utilizam a GPU de forma sequencial. Assim, para

baixo número de processos, a utilização de GPU apresenta desempenho superior devido a maior capacidade de processamento, no entanto, o aumento no número de processos eventualmente supera o desempenho da GPU. Vide Figura 13.

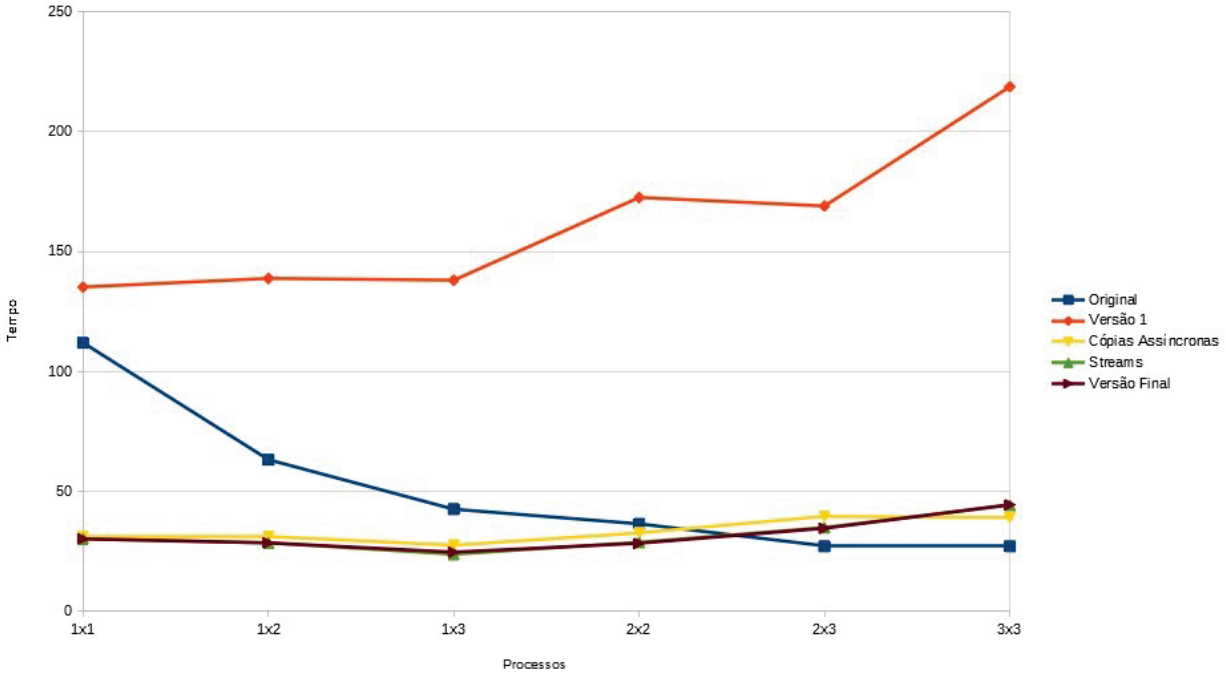

Figura 13. Visualização da tendência do tempo de execução de cada versão em função da quantidade de processos.

Na Figura 13 pode-se observar que, enquanto o tempo de execução da versão original tende a diminuir com o aumento do número de processos, o tempo de execução das versões utilizando processamento em GPU tente a aumentar. Desta forma, conforme o aumento do número de processos, melhor o desempenho da versão original em relação às versões utilizando GPU. Mais detalhes podem ser observados na Tabela 6.

Tabela 6. *Speedup* das versões com CUDA em relação à versão original do VTADV.

Processos	Versão final	Versão 1	Cópias assíncronas	Streams
1x1	3,7	0,83	3,59	3,7
1x2	2,22	0,46	2,03	2,22
1x3	1,73	0,31	1,55	1,79
2x2	1,28	0,21	1,12	1,27
2x3	0,78	0,16	0,68	0,77
3x3	0,57	0,12	0,65	0,57

Na Tabela 6 é mostrado o *speedup* das diferentes versões das sub-rotinas implementadas em relação à versão original. Como pode ser visto de forma gráfica Figuras 12 e 13, o desempenho da versão original em relação às implementadas com CUDA tende a melhorar conforme o aumento do número de processos.

Por exemplo, utilizando dois processos, a versão final com CUDA apresenta *speedup* de 3,7, indicando que o tempo de execução da sub-rotina foi reduzido em 3,7 vezes; porém, a mesma versão utilizando nove processos apresenta *speedup* de 0,57, indicando que a versão original executa em aproximadamente 57% da nova versão.

Considerando o melhor resultado de cada versão, duas das execuções com CUDA apresentam tempo menor que a versão original. Enquanto o melhor tempo para a versão original foi 25,33 segundos, utilizando nove processos, a versão final e a versão com streams apresentaram melhor desempenho utilizando três processos, com tempos de 24,65 segundos para a versão final e 23,76 segundos para a versão utilizando *streams*, representando um *speedup* de 1,03 e 1,07, respectivamente.

4.4 RESULTADOS DAS IMPLEMENTAÇÕES COM OPENACC

A implementação da sub-rotina com OpenACC foi estudada como uma alternativa ao uso de CUDA. A implementação de OpenACC ocorre de maneira muito mais rápida em relação a CUDA, porém a otimização do código é mais difícil, pois o programador não pode manipular diretamente o código executado pelo *device*.

Uma peculiaridade da versão OpenACC é a alta variação nos tempos de execução, enquanto a média do desvio padrão de todos os testes é de 1,74, o desvio padrão médio para as execuções com OpenACC é de 8,8. Para a execuções com dois e três processos, o desvio padrão da versão utilizando OpenACC é 14,18 e 21,71, respectivamente, enquanto para a versão final CUDA, o maior desvio padrão é de 1,55 para as execuções utilizando três processos.

Outra limitação quanto a utilização de OpenACC para a paralelização do modelo Eta foi em relação ao uso do *profiler*. Enquanto oficialmente OpenACC é compatível com NVVP, não foi possível a utilização do mesmo para analisar o modelo. Os resultados da implementacão utilizando OpenACC são apresentados na Tabela 7.

Tabela 7. Comparação entre o tempo médio de execução entre a versão original, a versão final com CUDA, e a versão com OpenACC.

Processos	Versão Original	Versão final CUDA	Versão OpenACC
1x1	111,97	30,12	93,06
1x2	63,22	28,51	74,65
1x3	42,55	24,65	115,70
2x2	36,45	28,37	97,25
2x3	27,04	34,60	147,67
3x3	25,32	44,43	166,65

Como pode ser observado na Tabela 7, o tempo médio obtido com OpenACC é maior para todos os casos quando comparado com a versão CUDA. No entanto, mesmo

apresentando performance inferior em relação a CUDA, a execução sequencial da versão OpenACC possui tempo inferior à versão original.

Comparado com as outras versões implementadas, a que mais se assemelha a versão com OpenACC é primeira versão com CUDA, sem a redução das cópias de memória, como pode ser visto na Figura 14.

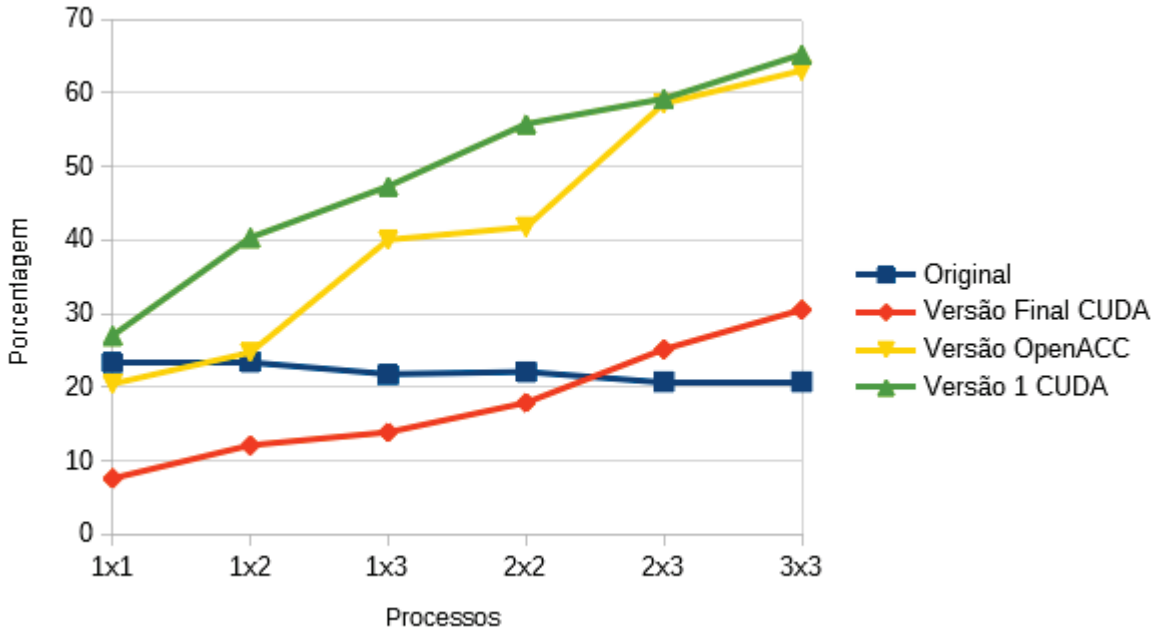

Figura 14. Porcentagem do tempo de execução do modelo Eta utilizado pela versão original, com OpenaCC, versão 1 com CUDA e versão final da suv-rotina VTADV com CUDA.

A Figura 14 representa a porcentagem do tempo total de execução do modelo Eta correspondente a cada versão da sub-rotina VTADV. Enquanto a porcentagem de tempo utilizada pela versão original se mantém relativamente constante, para as versões utilizando GPGPU a porcentagem aumenta com o número de processos. Isto se dá devido a competição pelo uso da GPU. Nesta Figura pode-se observar que, mesmo tentando otimizar as cópias de memória realizadas pela versão com OpenACC, o desempenho da mesma se assemelha ao da versão inicial com CUDA.

5. CONSIDERAÇÕES FINAIS

5.1 CONCLUSÃO

O presente trabalho teve como objetivo melhorar o desempenho do modelo Eta pelo uso de GPGPU, seguindo a linha de pesquisa iniciada por [32] e [7]. Neste trabalho foi utilizada uma nova versão do modelo Eta, disponibilizada pelo CPTEC/INPE, nesta versão a utilização de *common blocks* foi substituída pelo uso de *modules*, recurso compatível com computação paralela e distribuída.

Com base nessa nova versão foi criada uma versão heterogênea do modelo Eta, sobre a qual foram testados diversos recursos de GPGPU para obter ganho de desempenho. A região (quantidade de linhas) do código fonte paralelizada foi expandida significantemente em relação aos trabalhos anteriores, criando uma versão do VTADV com CUDA composta por 44 *kernels* e aproximadamente duas mil linhas de código, enquanto a original, com módulos possui aproximadamente mil, e também uma versão utilizando OpenACC.

Foram encontradas algumas dificuldades em relação a versão do modelo Eta sem *common blocks*, sendo necessário a adaptação de *scripts* da versão anterior para restaurar algumas funcionalidades do modelo, enquanto outros parâmetros não puderam ser alterados. Por exemplo, a versão disponibilizada não possibilitava a alteração do número de processos a ser utilizado, tal problema foi resolvido, no entanto, o tamanho da área de processamento não pode ser alterado, limitando o escopo dos testes realizados.

Enquanto a implementação de OpenACC não produziu resultados favoráveis, a integração da tecnologia CUDA ao modelo foi realizada com relativo sucesso. Quando executado para baixo número de processos, a parte do código paralelizada com CUDA apresenta um aumento significativo no desempenho, porém o desempenho decresce conforme o aumento do número de processos. Isto acontece devido a dificuldade de compartilhamento da GPU entre diferentes tarefas MPI, criando um gargalo na execução dos *kernels*.

Com os resultados obtidos é justificável dizer que a implementação de parte do modelo Eta com CUDA não é viável sob o ponto de vista de desempenho, uma vez que há muitas dependências em relação as cópias de memória entre as diferentes partes do código. Além disso, a competição pela utilização da GPU entre processos limita a escalabilidade do modelo, prejudicando o desempenho para grandes quantidades de processos, o que é uma necessidade para este tipo de aplicação. Tal problema pode potencialmente ser tratado em trabalhos futuros.

.

5.2 TRABALHOS FUTUROS

Para que a implementação do modelo utilizando MPI e CUDA seja expandida, primeiramente é necessário possibilitar a utilização concorrente da GPU entre diferentes tarefas MPI. Uma ferramenta que pode ser estudada para resolver tal problema é *Multi-Process Service* (MPS) [38]. MPS é uma implementação alternativa à API CUDA criada pela Nvidia para possibilitar o uso cooperativo da GPU em aplicações CUDA com múltiplos processos.

Apesar de apresentar tempo de execução superior em relação as outras versões, OpenACC pode ser implementado de forma muito mais ampla e em menor tempo que CUDA, diversos estudos indicam que OpenACC é capaz de obter desempenho próximo à CUDA. Caso seja conseguido obter um tempo similar à CUDA utilizando OpenACC, este pode se tornar uma alternativa viável.

Estudos anteriores, como o de [7], apontam dificuldades no uso de OpenMP em aplicações Fortran com *common blocks*, evidenciados por resquícios de código OpenMP inutilizado presente em algumas partes do modelo. Utilizando esta nova versão do modelo Eta, onde os mesmos foram substituídos por módulos, é possível uma implementação utilizando OpenMP e MPI de forma conjunta.

Outro trabalho possível é a remoção de MPI, criando uma versão do modelo Eta para ser utilizada em apenas um computador com uma GPU. Com a remoção de MPI pode-se criar uma versão do modelo completamente em CUDA. Outra opção é a implementação de uma versão multi-core, composta de CUDA e OpenMP, dividindo o processamento do modelo entre a GPU e os núcleos da CPU, dependendo de qual se aplica melhor à cada tarefa. Outro benefício do uso de OpenMP é que o mesmo não possui o mesmo problema encontrado com MPI, sendo possível a utilização concorrente da GPU por diferentes processos.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Stough III, H. P.; WATSON, J. F.; JARRELL, M. A. New technologies for reducing aviation weather-related accidents. 2006. Disponível em:<https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20060048302.pdf>. Acesso em: 03 Jun. 2020.

[2] RODRIGUEZ, D. et al. Predicting optimum crop designs using crop models and seasonal climate forecasts. Scientific Reports, v. 8, n. 1, p. 2231, 2018. ISSN 2045-2322. Disponível em: <https://doi.org/10.1038/s41598-018-20628-2>.

[3] LUXEMBOURG, U. du. High Performance Computing. 2020. Disponível em: <https://wwwen.uni.lu/university/high_performance_computing>. Acesso em: 26 Fev. 2020.

[4] KALNAY, E. Atmospheric Modeling, Data Assimilation and Predictability. The Old Schools, Trinity Ln, Cambridge CB2 1TN, United Kingdom: Cambridge University Press, 2002.

[5] CHOU, S. C. et al. Evaluation of the Eta Simulations Nested in Three Global Climate Models. American Journal of Climate Change, Scientific Research Publishing, v. 03, n. 05, p. 438–454, dez. 2014. ISSN 2167-9495. Disponível em: <http://www.scirp.org/journal/PaperInformation.aspx?PaperID=52877>. Acesso em: 21 Out. 2018.

[6] CHAN, C. S. Modelo Eta: Desenvolvimento e Planos. 2016. Disponível em: <ftp://ftp1.cptec.inpe.br/etamdl/Workshop/WorkEtaV/Apresenta%C3%A7%C3%B5es/>. Acesso em: 11 Abr. 2020.

[7] FLORES, H. G. Integração da tecnologia CUDA ao modelo de previsão do tempo Eta. Dissertação (Mestrado) — PPGCA - Universidade de Passo Fundo, Av. Brasil Leste, 285 - São José, Passo Fundo - RS, 99052-900, 2018.

[8] GRANDE, U. F. de C. Modelagem Numérica da Atmosfera. 2013. Disponívelem: <http://www.dca.ufcg.edu.br/mna/Anexo-MNA-modulo02a.pdf>. Acesso em: 11 Abr. 2020.

[9] FIGUEROA, S. N. Brazilian global atmospheric model (bam). In: 1º Workshop de Inovação do INPE. São José dos Campos -SP, Brasil: INPE, 2016.

[10] MODEL | Eta Model. Disponível em: <http://etamodel.cptec.inpe.br/history/>. Acesso em: 21 Out. 2018.

[11] MESINGER, F. et al. The step-mountain coordinate: Model description and performance for cases of alpine lee cyclogenesis and for a case of an appalachian redevelopment. Monthly Weather Review, American Meteorological Society,

v. 116, n. 7, p. 1493–1518, jul 1988. Disponível em: <https://doi.org/10.1175/1520-0493(1988)116<1493:tsmcmd>2.0.co;2>.

[12] CHOU, S. C. MODELO REGIONAL ETA. Disponível em:<http://climanalise.cptec.inpe.br/~rclimanl/boletim/cliesp10a/27.html>. Acesso em: 21 Out. 2018.

[13] JORGENSEN, E. Introduction to Programming using Fortran 95/2003/2008. 2018. Disponível em: <http://www.egr.unlv.edu/~ed/fortranv3.pdf>. Acesso em: 21 Out. 2018.

[14] FLORES, H. G.; MELLO, A. L.; REBONATTO, M. T. Introdução à programação paralela em fortran com open-mp e mpi. In: SCHNORR, L. M.; PILLON, M. A. (Ed.). ERAD 2018: 18ª Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul. Porto Alegre, RS, Brasil: Sociedade Brasileira de Computação – SBC, 2018. cap. 4, p. 27–43.

[15] PGI. PGI | About. Disponível em: <https://www.pgroup.com/about/>. Acesso em: 21 Out. 2018.

[16] HARPER, D.; STOCKMAN, L. M. FORTRAN 77 - COMMON Blocks, BLOCK DATA and EQUIVALENCE. Disponível em: <https://www.obliquity.com/computer/fortran/common.html>. Acesso em: 21 Out. 2018.

[17] GOMES, J. L. Re: Eta sem common blocks [mensagem pessoal]. 2018. Mensagem recebida por a.lexii@hotmail.com. Acesso em: 5 Set. 2018.

[18] MIKUSHIN, D. et al. Generic approach to Legacy Fortran code porting on GPU. Generic approach to Legacy Fortran code porting on GPU. Oct 2015. Disponível em: <https://cds.cern.ch/record/2062033>.

[19] OWENS, J. et al. Gpu computing. Proceedings of the IEEE, v. 96, p. 879–899, 05 2008.

[20] NVIDIA. Programming Guide :: CUDA Toolkit Documentation. 2020. Disponível em: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/#from-graphics-processing-to-general-purpose-parallel-computing >. Acesso em: 26 Fev. 2020.

[21] MICHALAKES, J.; VACHHARAJANI, M. Gpu acceleration of numerical weather prediction. Parallel Processing Letters, World Scientific Pub Co Pte Lt, v. 18, n. 4, p. 531–548, dez 2008. Disponível em: <http://dx.doi.org/10.1142/S0129626408003557>.

[22] RUETSCH, G.; PHILLIPS, E.; FATICA, M. Gpu acceleration of the long-wave rapid radiative transfer model in wrf using cuda fortran. In: Many-Core and Reconfigurable Supercomputing Conference. Roma, Itália: MSRC, 2010.

[23] Mielikainen, J. et al. Gpu acceleration of the updated goddard shortwave radiation scheme in the weather research and forecasting (wrf) model. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, v. 5, n. 2, p. 555–562, 2012.

[24] Vanderbauwhede, W.; Takemi, T. An investigation into the feasibility and benefits of gpu/multicore acceleration of the weather research and forecasting model. In: 2013 International Conference on High Performance Computing Simulation (HPCS). Helsinque, Finlândia: HPCS, 2013. p. 482–489.

[25] MüLLER, M.; AOKI, T. New high performance gpgpu code transformation framework applied to large production weather prediction code. ACM Transactions on Parallel Computing, v. 5, 02 2018.

[26] OPENACC. About OpenACC. Disponível em: <https://www.openacc.org/about>. Acesso em: 21 Out. 2018.

[27] LARKIN, J. OPENACC ONLINE COURSE 2018. Nvidia, 2018. Disponível em: <http://info.nvidia.com/rs/156-OFN-742/images/OpenACC%20Course%202018%20Week%201.pdf>. Acesso em: 21 Out. 2018.

[28] Nvidia Corporation. OpenACC. 2015. Disponível em: <https://www.pgroup.com/lit/brochures/openacc_sc15.pdf>. Acesso em: 22 Out. 2018.

[29] HOSHINO, T. et al. Cuda vs openacc: Performance case studies with kernel benchmarks and a memory-bound cfd application. In: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing. Delft, Países Baixos: IEEE press, 2013. p. 136–143.

[30] NVIDIA. Programming Guide :: CUDA Toolkit Documentation. 2020. Disponível em: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/#thread-hierarchy>. Acesso em: 26 Fev. 2020.

[31] NVIDIA. Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Fermi. [S.l.], 2009. Disponível em: <https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf>. Acesso em: 21 Out. 2018.

[32] MELLO, A. L. Alternativa à paralelização do modelo Eta: um estudo de caso com CUDA. Dissertação (Bacharelado) — Ciência da Computação - Universidade de Passo Fundo, Av. Brasil Leste, 285 - São José, Passo Fundo - RS, 99052-900, 2016.

[33] GOMES, J. L. Re: Erro Modelo Eta [mensagem pessoal]. 2019. Mensagem recebida por a.lexii@hotmail.com. Acesso em: 08 Jan. 2019.

[34] NVIDIA. Programming Guide :: CUDA Toolkit Documentation. 2020. Disponível em: <https://docs.nvidia.com/cuda/profiler-users-guide/index.html#visual>. Acesso em: 26 Fev. 2020.

[35] RUETSCH, G.; FATICA, M. CUDA Fortran for Scientists and Engineers: Best Practices for Efficient CUDA Fortran Programming. Amsterdã, Países Baixos: Elsevier Science, 2013. 338 p. ISBN 9780124169722.

[36] NVIDIA. GPU Pro Tip: CUDA 7 Streams Simplify Concurrency. 2015. Disponível em: <https://devblogs.nvidia.com/gpu-pro-tip-cuda-7-streams-simplify-concurrency/>. Acesso em: 26 Fev. 2020.

[37] LUITJENS, J. Cuda streams: Best practices and common pitfalls. In: . Califórnia, Estados Unidos da América: GTC Silicon Valley, 2014.

[38] NVIDIA. Multi-Process Service. 2019. Disponível em: <https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf>. Acesso em: 18 Fev. 2020.

APÊNDICE A – RESULTADOS DOS TESTES

Processos	Original	Versão final	OpenACC	Versão 1	Cópias assíncronas	Streams
1x1	111.6796	30.17015	93.06038	131.6898	31.13958	30.25034
1x1	111.7001	30.25047	93.20089	134.4407	31.17041	30.16029
1x1	111.8805	30.18025	93.22018	130.6503	31.0401	30.3098
1x1	111.2899	30.22027	93.14056	130.6699	30.82038	30.02062
1x1	111.3906	30.14967	93.25964	131.0302	31.29969	30.20036
1x1	111.1703	30.05052	92.65026	136.0604	30.92031	30.00994
1x1	111.5299	30.14982	93.19965	134.0401	31.41936	30.24985
1x1	111.7402	29.89023	92.90960	135.2604	30.9599	30.1998
1x1	111.2905	30.08993	93.17988	134.4201	31.17008	30.12
1x1	111.7304	29.96007	92.99045	138.1606	31.16092	30.25958
1x1	111.8398	30.32963	93.09058	139.9809	31.06	29.99037
1x1	112.2509	30.20039	93.22025	135.0801	31.2596	30.32024
1x1	111.3197	30.32095	93.04025	137.2399	31.23056	30.19983
1x1	111.4002	30.27986	93.40029	135.11	31.07031	30.30065
1x1	112.1615	30.36046	93.06995	132.0904	31.12037	30.30999
1x1	112.0001	30.27065	93.04061	140.0601	31.43035	30.4202
1x1	111.72	30.39048	92.91010	130.5695	30.87039	29.98032
1x1	111.5702	29.98044	93.33984	130.89	31.06024	30.4996
1x1	111.47	30.00932	93.13015	139.2496	31.08023	30.03969
1x1	111.5307	30.15991	93.17034	130.4803	31.30026	30.31021
1x1	111.3991	30.34001	92.99043	130.5302	31.09948	30.5599
1x1	111.7817	29.89003	92.91056	130.4804	31.04013	30.1204
1x1	111.99	30.0698	93.04077	139.6302	31.42017	30.25044
1x1	112.2203	29.98049	92.67930	134.0803	31.25017	30.46032
1x1	111.6801	30.16029	93.17040	135.3906	30.86019	30.08974
1x1	112.3403	30.14986	93.28043	136.0101	31.27043	29.97991
1x1	111.8901	30.1202	93.00042	140.2504	31.17006	30.32016
1x1	111.4714	30.43972	93.11987	135.2599	30.95978	30.18974
1x1	112.2299	29.99027	93.12061	130.5301	31.20054	30.26991
1x1	111.82	30.25025	92.94990	135.22	30.97012	30.13054
1x1	111.7498	30.22021	92.82001	139.4996	30.98037	30.33976
1x1	111.6106	30.29025	93.26060	139.5103	31.21987	30.16002
1x1	111.8202	30.22034	93.15994	138.4105	31.16969	30.35995
1x1	111.7995	30.36025	93.17021	132.1604	30.94005	30.22994

Processos	Original	Versão final	OpenACC	Versão 1	Cópias assíncronas	Streams
1x1	111.2202	30.34013	92.76069	141.5704	31.09991	30.32014
1x1	111.6307	30.3096	92.58009	135.6195	31.05996	30.22003
1x1	112.2702	30.28036	93.25012	135.0601	30.97033	29.94001
1x1	111.6499	30.28025	92.94948	134.9306	31.10014	30.22985
1x1	112.2002	30.19951	92.78043	139.7799	31.34979	30.18969
1x1	111.5203	30.22971	93.01093	139.7004	30.98041	30.1504
1x2	63.68006	28.72002	95.44013	138.9506	31.13037	27.94023
1x2	64.00012	28.19987	94.63075	138.6705	31.13007	27.96968
1x2	63.14993	28.38025	60.82957	139.7599	31.43029	28.9001
1x2	63.02995	28.18981	84.40039	139.1406	31.10034	28.3801
1x2	63.39003	28.52983	63.12959	139.0299	31.03988	28.51998
1x2	63.54	28.47016	64.10045	138.4003	30.96002	28.69
1x2	62.85014	28.46981	63.32042	140.1603	31.30019	28.22017
1x2	62.91027	29.17005	65.34013	139.1999	31.29006	27.84022
1x2	63.63011	28.32994	62.07979	138.6396	30.93015	28.69032
1x2	63.03019	28.72001	93.15989	140.37	31.12017	28.6003
1x2	62.73983	28.49038	76.44975	139.1902	31.30003	28.05036
1x2	63.59006	28.33998	60.98042	139.6203	31.1603	28.27012
1x2	63.31006	28.70005	63.89003	136.7202	31.23005	28.33002
1x2	62.82004	28.56992	63.74995	138.31	31.3198	28.96982
1x2	63.56989	28.24985	92.93068	138.3397	31.14987	28.47975
1x2	62.91001	28.2301	62.65997	138.5193	31.30003	28.78021
1x2	63.27006	28.13987	63.03014	137.9102	30.92011	28.81027
1x2	64.02999	28.72017	95.26025	137.5005	30.97025	28.73013
1x2	63.06998	28.57006	63.30023	138.5004	31.2801	28.27044
1x2	62.93014	28.85003	62.07977	137.8094	31.23007	28.57967
1x2	63.58012	28.14048	61.42965	137.1606	31.18005	28.70002
1x2	63.36012	28.31026	65.10035	140.3009	31.16002	28.66035
1x2	62.9398	28.80982	67.14003	138.7699	31.03974	28.62999
1x2	63.94011	28.03015	64.69025	138.2702	30.94004	28.79993
1x2	62.98024	28.49951	61.26016	140.3306	30.71014	28.86039
1x2	63.72003	29.03995	90.03980	139.1595	31.08031	29.06992
1x2	62.94018	29.38	66.83002	139.6005	31.15003	28.3905
1x2	63.8502	28.36985	93.51003	138.0003	31.10995	28.3304
1x2	62.78033	28.89989	62.65975	138.9409	30.95028	28.83008
1x2	63.07013	28.35036	90.77019	139.0096	30.97016	28.82018
1x2	62.8702	27.93036	64.03018	139.08	31.32982	28.29979
1x2	63.1102	28.76011	95.14975	140.2202	30.90006	28.11043

Processos	Original	Versão final	OpenACC	Versão 1	Cópias assíncronas	Streams
1x2	63.18023	28.42015	96.02975	138.8094	31.23005	28.37012
1x2	63.84018	28.74998	88.22006	140.5205	30.94008	28.94023
1x2	62.88003	28.20026	62.49022	138.4301	31.23015	28.9098
1x2	63.02026	27.83986	62.45968	138.67	31.04977	28.59001
1x2	62.79018	28.90971	64.95961	138.9197	31.19003	28.65028
1x2	62.80983	28.52003	93.24970	137.7605	31.22999	28.40053
1x2	62.82999	28.50027	93.37016	138.8198	31.03002	28.32023
1x2	62.70005	28.75997	92.02972	138.3101	31.09031	28.83016
1x3	43.2498	25.11033	139.2605	139.2101	27.59017	24.58005
1x3	42.5601	22.02998	81.95038	139.0801	28.04996	25.98006
1x3	41.93966	20.75992	106.0597	135.68	26.94024	26.13995
1x3	42.3504	25.76003	126.7303	140.7795	27.65989	21.77011
1x3	43.03918	25.43003	139.8197	142.1106	27.44991	26.5503
1x3	43.39003	24.30006	86.25027	142.1402	27.9202	25.99989
1x3	42.18028	25.01979	77.94989	139.9501	27.49992	22.9598
1x3	42.24979	25.33021	138.9297	138.5	27.57027	21.93009
1x3	42.93991	25.44019	119.7599	142.9202	27.99017	21.67997
1x3	41.73014	24.53983	126.5401	135.8902	27.74987	25.08001
1x3	42.37003	21.90972	76.39014	140.6496	27.57013	23.8503
1x3	43.01949	25.76013	139.9898	136.3803	27.42003	26.18002
1x3	41.77017	24.76028	129.7601	136.1399	27.00006	25.32
1x3	42.37027	25.37968	139.7303	142.9903	27.4698	25.80995
1x3	42.62003	26.37026	80.27013	131.8507	27.34004	25.22014
1x3	41.89006	24.68	114.9700	142.3904	27.39043	21.16034
1x3	42.26987	25.37012	123.0403	141.7005	27.69026	26.41993
1x3	43.13985	25.93002	130.7104	134.1402	27.35027	21.79025
1x3	42.5601	21.51059	79.97987	140.6497	27.26052	25.87995
1x3	42.11956	25.71025	113.4101	131.8699	27.36981	20.48007
1x3	43.10999	25.13009	105.9501	142.9699	28.45022	26.23027
1x3	43.08019	24.2101	119.1501	129.2304	27.57032	21.59006
1x3	41.90014	23.11006	128.5900	134.7995	27.91039	21.15029
1x3	43.01019	26.12964	136.6308	129.9799	27.58045	24.13988
1x3	42.49031	25.56051	111.6400	138.9901	27.98031	25.32027
1x3	43.2402	24.84973	138.7098	134.69	26.70976	21.68
1x3	42.49998	26.19993	130.8996	141.3198	27.03027	22.38032
1x3	43.09999	26.10997	126.9401	140.0801	26.72015	21.47005
1x3	44.36062	26.19016	91.44042	138.9195	27.10028	21.7603
1x3	42.29032	23.49005	86.79951	141.4901	27.92992	23.90989

Processos	Original	Versão final	OpenACC	Versão 1	Cópias assíncronas	Streams
1x3	41.81019	24.80962	78.16020	129.1098	26.98979	21.63021
1x3	42.62983	21.70964	122.2803	131.76	28.17013	24.38036
1x3	42.24009	25.01027	139.2501	134.1603	27.81024	20.76
1x3	43.05997	21.36003	104.4599	142.2105	27.9	22.5301
1x3	42.56046	25.06048	131.6300	143.3605	27.08996	24.77982
1x3	42.32949	26.1802	138.9008	142.2804	26.9701	26.01988
1x3	43.03989	26.27989	132.3501	136.92	28.2003	21.29991
1x3	41.63011	24.77016	80.96004	140.2997	28.40977	25.2097
1x3	41.87004	22.67046	123.7303	140.1097	27.15004	24.00956
1x3	42.05988	26.14987	128.2600	136.4996	27.58012	25.36007
2x2	36.69023	27.87982	106.7603	175.6295	32.51009	29.5102
2x2	36.94007	30.05014	106.3799	171.6104	32.51998	27.56985
2x2	35.89016	27.22968	85.18018	170.8599	32.34999	29.92001
2x2	35.99997	27.55019	105.7300	175.9601	32.57986	30.07995
2x2	36.82003	29.76007	106.4401	177.79	32.83008	29.95996
2x2	36.70006	29.32047	106.0600	164.1402	32.86051	29.75006
2x2	36.21002	27.24998	79.46025	165.8303	32.99007	29.99032
2x2	36.50006	27.61015	106.2902	171.2101	32.98005	29.92014
2x2	36.75005	27.23978	106.6303	172.8302	32.18006	28.47011
2x2	36.32995	29.23009	78.70028	169.2201	32.77987	30.08015
2x2	36.70028	25.82988	106.1200	171.9601	32.33027	29.83979
2x2	36.7602	27.63016	97.28975	170.6395	32.84992	28.63049
2x2	36.56993	29.79023	77.98028	168.6302	32.8301	30.03995
2x2	36.48022	29.8102	78.89026	171.0404	32.79977	27.34975
2x2	36.60018	29.46986	77.13017	173.6297	32.85991	26.72013
2x2	36.38004	26.50998	96.00023	173.91	32.62013	30.06987
2x2	35.95991	26.7598	106.3098	171.8105	32.01992	27.44996
2x2	36.94997	27.7801	106.7301	174.6098	32.37016	28.75006
2x2	36.34002	29.03993	76.17005	178.0805	32.51982	30.12024
2x2	36.45	29.82001	106.7303	164.83	32.5004	27.44993
2x2	36.67991	29.55004	106.6703	171.5803	32.67014	27.58031
2x2	36.73993	26.84973	106.3402	171.4203	32.61002	28.93012
2x2	36.03999	30.09013	105.1904	172.03	33.38	27.38014
2x2	36.86991	29.14015	77.86992	178.4207	32.83998	28.70995
2x2	36.18993	29.98044	106.7407	170.9202	32.78999	30.31998
2x2	36.78006	28.95996	106.9200	170.8402	32.96989	29.86036
2x2	36.00992	29.87998	103.7102	172.1003	32.35004	25.89991
2x2	36.83013	29.75983	100.6499	167.4808	32.47005	26.53995

Processos	Original	Versão final	OpenACC	Versão 1	Cópias assíncronas	Streams
2x2	36.5699	29.65007	106.0900	177.8104	32.81007	28.00017
2x2	36.27001	27.9002	84.94018	175.7902	32.8298	30.06996
2x2	36.34986	27.27997	106.6303	178.2899	32.56007	29.36986
2x2	36.11016	27.68998	106.2601	183.5801	32.66997	30.16981
2x2	36.26982	27.50029	82.27984	178.5802	32.56001	30.03008
2x2	36.86018	29.53009	94.52016	173.9605	32.60996	29.99982
2x2	36.22018	27.08995	106.4504	170.0304	32.67025	30.12001
2x2	36.26984	30.01027	76.44056	170.0505	33.01006	25.63001
2x2	36.18008	29.70029	93.51991	174.8304	33.00019	29.80009
2x2	35.89995	26.67005	106.5499	175.8698	32.40996	25.76003
2x2	36.45979	26.05012	95.83987	174.0201	32.83001	26.75991
2x2	36.40034	26.02001	99.39981	162.2105	32.84013	28.81005
2x3	28.07992	32.5602	150.9204	167.8702	39.76013	35.49009
2x3	28.16998	35.08009	140.2503	172.7402	39.77002	34.59008
2x3	26.51988	35.36009	151.0099	168.6898	39.63008	34.55008
2x3	26.36995	34.74005	151.3200	173.8698	39.22032	35.39994
2x3	27.55005	33.01006	142.9500	166.9103	39.51001	36.04002
2x3	27.6901	32.90004	150.5698	166.0601	39.6002	35.63978
2x3	26.3301	35.86018	140.8000	165.5405	39.74015	35.27017
2x3	26.23003	35.28994	149.1097	171.85	39.64012	34.58016
2x3	28.07998	34.93998	151.8301	163.78	39.55007	36.26015
2x3	26.48024	34.94022	151.6001	176.5104	39.65016	34.55007
2x3	26.4101	34.17995	149.2405	170.4403	39.61007	35.10002
2x3	28.09014	34.98991	147.0201	163.9106	39.05008	35.5497
2x3	26.18012	33.66995	141.5602	182.9901	39.56015	35.38007
2x3	26.22011	35.85009	149.8398	175.3601	40.08008	33.37007
2x3	27.47014	33.41997	141.5496	162.6998	39.53001	35.62996
2x3	26.77008	34.83991	136.8599	167.2997	39.65013	34.75014
2x3	26.54014	35.13012	150.3003	171.3902	39.55993	35.76989
2x3	28.17993	32.79015	151.0399	161.5202	39.84981	32.60006
2x3	26.08998	34.28011	149.0498	157.3802	39.67019	34.07007
2x3	26.56019	35.54013	146.0305	169.9198	39.73002	35.23016
2x3	28.41002	35.36008	151.4103	168.0704	39.3601	36.19012
2x3	27.6201	35.60027	144.9601	177.9996	39.23018	34.43999
2x3	26.39969	35.46997	151.9502	175.12	39.55	35.25009
2x3	28.18024	34.73997	151.8601	172.2803	39.49014	34.92007
2x3	26.38	34.03003	135.2199	170.6302	39.24004	34.15002
2x3	28.37998	34.41997	141.6402	179.3602	39.53019	35.20012

Processos	Original	Versão final	OpenACC	Versão 1	Cópias assíncronas	Streams
2x3	27.03002	34.78006	147.9000	160.6404	39.96001	35.5301
2x3	28.29019	32.31002	151.8403	166.45	39.47013	33.96968
2x3	26.68012	35.66001	147.6094	171.5701	39.91006	33.74006
2x3	27.24013	33.92011	147.3800	171.8703	39.19985	35.45994
2x3	26.20023	35.18993	147.3299	162.5803	39.43996	33.88996
2x3	26.41004	35.83023	139.2999	164.8399	39.04	33.81981
2x3	27.28009	34.54993	149.2402	165.2902	39.44999	35.6399
2x3	27.47006	35.66004	151.7305	169.9001	39.31991	34.98004
2x3	26.29005	33.36998	149.4603	170.1499	39.54996	34.39003
2x3	26.43015	35.78005	150.9802	176.3801	39.70978	34.68996
2x3	27.97995	33.96012	152.4602	166.1403	39.98009	35.36012
2x3	25.63016	33.35001	151.7399	163.9601	39.83011	34.32995
2x3	26.53007	35.72008	149.0703	169.4501	39.62004	35.29017
2x3	26.72998	34.90994	150.7300	162.7602	39.83995	36.0599
3x3	25.10025	44.10996	167.0098	219.6901	40.32998	44.70021
3x3	25.55013	44.19999	167.0299	226.0897	40.13008	44.00014
3x3	26.45001	44.14011	166.63	218.7196	38.14002	44.58004
3x3	26.26002	44.77024	166.11	215.7303	38.65012	44.72004
3x3	25.86003	44.60004	167.1097	212.9408	38.78997	44.2601
3x3	25.63016	43.57019	166.8999	216.7201	39.15981	44.17007
3x3	25.22013	44.6102	167.2502	219.7706	39.08009	43.87991
3x3	25.15983	44.53994	166.6303	215.4702	39.75011	44.62011
3x3	25.23993	43.89025	166.9598	221.2899	38.71014	43.3901
3x3	24.95003	44.74002	166.4601	224.5508	39.23999	44.78003
3x3	24.86992	44.15011	165.5399	218.4599	39.18995	44.45006
3x3	25.77014	44.54011	166.3298	218.1299	38.80011	44.45001
3x3	25.16027	44.44996	166.9403	222.5799	38.82001	43.19999
3x3	25.43993	44.31993	166.8599	219.2303	38.55989	44.05016
3x3	24.65989	43.84998	167.1602	216.84	38.93002	44.06012
3x3	24.73992	44.48998	167.3501	218.1504	38.85025	44.51997
3x3	25.34008	43.26994	166.5999	217.3902	37.77018	44.18019
3x3	25.24994	44.3097	166.7798	215.3002	39.18021	44.12014
3x3	24.99989	44.64009	165.9299	220.1403	38.6201	43.70018
3x3	24.80994	45.21009	166.6093	218.5498	38.40986	44.22003
3x3	24.97011	44.79022	166.8902	212.7504	38.99996	43.90007
3x3	25.74004	44.06976	166.7402	222.8704	39.15987	44.8002
3x3	26.16003	44.66017	165.2302	222.64	39.32014	44.70001
3x3	24.9399	43.97991	166.9702	213.1707	38.77	43.84001

Processos	Original	Versão final	OpenACC	Versão 1	Cópias assíncronas	Streams
3x3	25.15008	44.63029	166.5204	218.2201	38.59	44.10982
3x3	25.12013	44.75998	167.0402	222.7603	38.9599	44.88002
3x3	25.00017	44.5502	166.0897	215.4298	38.90015	44.21986
3x3	24.70027	44.69997	167.0498	216.8908	39.23017	44.18023
3x3	25.90007	44.37006	166.97	221.5702	40.05009	44.28986
3x3	25.16001	44.66994	166.4901	216.5103	39.50011	44.09996
3x3	25.52002	45.01998	167.24	223.6302	39.26005	44.41997
3x3	24.46012	44.61006	166.5999	221.8094	38.71989	44.65988
3x3	25.15037	44.19987	167.0604	218.1403	38.82991	44.82992
3x3	25.54006	44.4601	166.1303	215.8002	40.12998	44.07004
3x3	25.07012	44.37015	165.4498	221.2598	39.15027	44.84002
3x3	26.04023	44.25999	166.5301	217.6499	39.00999	43.82008
3x3	25.67004	44.5601	166.7	216.4406	38.91985	44.43005
3x3	25.44007	44.79008	166.5004	223.1506	38.52991	44.92001
3x3	25.51016	44.80999	166.4206	220.2894	38.69004	44.4301
3x3	25.32015	44.62007	167.1998	216.4408	39.26023	44.28014

APÊNDICE B – ANÁLISE ESTATÍSTICA

Tabela 1. Desvio padrão das repetições para cada teste

Processos	Original	Versão final	OpenACC	Versão 1	Cópias assíncronas	Streams
1x1	0,3098	0,1377	0,1875	3,4404	0,154	0,1428
1x2	0,3919	0,3261	14,1843	0,8735	0,1493	0,3054
1x3	0,569	1,5481	21,7101	4,1154	0,4325	1,9605
2x2	0,3047	1,3499	11,6032	4,2957	0,2581	1,4294
2x3	0,7994	1,003	4,5966	5,4609	0,2396	0,8051
3x3	0,4497	0,3748	0,4915	3,216	0,5162	0,3978

Tabela 2. Resultados dos Testes t realizados. Diferença entre os tempos é significativa quando valor-p é menor que 0.05.

Processos	Original e Final	Final e Cópias Assíncronas	Final e Streams
1x1	4,28167419792783E-126	3,21458961630873E-42	0,377532385499503
1x2	2,4727696338247E-129	4,28336718343804E-45	0,708427013858652
1x3	2,30236308434903E-50	1,25557968273286E-14	0,028737391689326
2x2	5,93646855634045E-34	1,11290199218753E-22	0,19265102082021
2x3	1,85836940607259E-49	9,32766042128631E-31	0,114972944533124
3x3	2,66336630964653E-105	6,85700234543383E-59	0,119536539588046

Tabela 3. Média dos tempos obtidos.

Processos	Original	Versão final	OpenACC	Versão 1	Cópias assíncronas	Streams
1x1	111.72	30.19	93.06	135.27	31.12	30.22
1x2	63.22	28.51	74.65	138.85	31.12	28.54
1x3	42.55	24.65	115.71	138.11	27.54	23.76
2x2	36.45	28.37	97.25	172.6	32.68	28.79
2x3	27.04	34.6	147.67	169.05	39.58	34.93
3x3	25.33	44.43	166.65	218.83	39.03	44.29

UPF
UNIVERSIDADE
DE PASSO FUNDO

UPF Campus I - BR 285, São José
Passo Fundo - RS - CEP: 99052-900
(54) 3316 7000 - www.upf.br