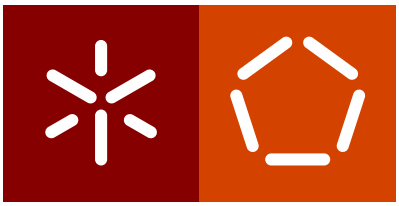
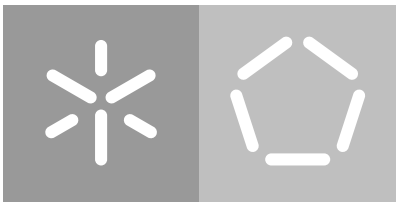


Universidade do Minho
Escola de Engenharia
Departamento de Eletrónica Industrial

Bruno Miguel Freitas Ribeiro

Otimização do Código de Remoção de Imagem Fantasma Utilizado na Missão Espacial Proba-3

Agosto de 2020



Universidade do Minho
Escola de Engenharia
Departamento de Eletrónica Industrial

Bruno Miguel Freitas Ribeiro

Otimização do Código de Remoção de Imagem Fantasma Utilizado na Missão Espacial Proba-3

Dissertação de Mestrado
Mestrado Integrado em Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Fernando Ribeiro

Agosto de 2020

AGRADECIMENTOS

A realização desta dissertação resultou de muito trabalho e horas de dedicação que culminaram no cumprimento de todo o trabalho proposto, ao qual não posso deixar de agradecer a todas as pessoas que fizeram com que tudo isto fosse possível.

Em primeiro lugar ao meu orientador Doutor Fernando Ribeiro, por todo o apoio, toda a confiança em mim depositada, pelo boa disposição e profissionalismo com que me recebeu ao longo de todos estes anos e principalmente durante a realização desta dissertação.

Um enorme agradecimento ao Doutor Dalmiro Maia, sem ele esta dissertação não seria possível. Agradeço desde já toda a confiança depositada em mim para a realização deste projeto aliciante que tanto me deu prazer em explorá-lo.

Um agradecimento muito especial aos meus pais Carmo Freitas e João Ribeiro, por todo o esforço e dedicação que fizeram por mim ao longos destes cinco anos de curso, tendo sempre me demonstrado um apoio incondicional ao longo de todo este trajeto.

Um grande agradecimento ao meu irmão Hélder Ribeiro, que esteve sempre presente ao longo de todo o meu percurso académico, ajudando-me em tudo que precisei desde o início até ao fim desta aventura.

Um enorme agradecimento à minha namorada, Ana Araújo, por todo o incentivo, apoio, compreensão, lealdade e por ter sido a minha fonte de inspiração nos momentos mais difíceis deste trajeto.

Por fim, e não menos importante, quero agradecer à minha família todo o apoio demonstrado ao longo destes cinco anos, todos as conquistas partilhadas lado a lado, todos os obstáculos superados, e por todo o amor incondicional demonstrado e vivido durante todo este tempo.

A todos, um muito obrigado!

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração. Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

ABSTRACT

This dissertation appeared in the scope of the PROBA-3 space mission from *ESA*. This mission aims to study in more detail the solar corona, with the objective of getting important data about the sun and the sun rays. These observation will be done by two satellites in a formation flight. This two satellites will form a coronagraph, which is an instrument used to perform observations to the solar corona. Despite this instrument already eliminates part of the luminous noise originated by other light sources, it is necessary that this noise is also reduced by software methods. Due to the misalignments caused by the formation flight of the two satellites, the image obtained needs to be processed pixel by pixel, thus increasing the execution time of the code.

This dissertation aims to study and implement methods that optimize the code already developed, so that it has the ability to analyze more images in a shorter period of time.

The optimization method that will have the greatest focus on this dissertation is parallelization. The initial code will initially be parallelized with OpenACC, this being a programming model that allows the code to be paralyzed only with the addition of directives. With this model, the performance of the execution of the parallel code for CPU and for GPU will be studied so that both performances can be compared.

Because communication and memory transfer between *CPU* and *GPU* takes a long time, running parallel code on *GPU* may not be effective if there is a need for multiple memory transfers between the two processors. Although OpenACC allows the code to be parallelized without changing it, some parts of the code in order to be parallelized need to be modified. Due to having parts of the code to be executed in *CPU* and others in *GPU*, it is necessary to transfer memory between them, thus making the code performance worse. To obtain complete parallelization of the code in order to eliminate the time spent on data transfer, the code was changed and coded in *CUDA*. *CUDA* is an NVIDIA framework that allows to write code to run on NVIDIA's *GPU*. Using *CUDA* it is possible to analyze the true potential of a *GPU* in solving mathematical problems.

Palavras chave: PROBA-3, OpenACC, Parallelization, CPU, GPU, CUDA, Optimization

RESUMO

Esta dissertação surgiu no âmbito da missão espacial PROBA-3 da *ESA*. Esta missão tem como objetivo estudar em maior profundidade a coroa solar, com o intuito de obter dados importantes sobre o sol e os raios solares. Para realizar esta observação irão ser lançados dois satélites que irão realizar um trajeto em formação. Estes dois satélites irão formar um *coronagraph*, um instrumento utilizado para observar a coroa solar. Apesar deste instrumento já eliminar parte do ruído luminoso originado por outras fontes de luz é necessário que este ruído seja também reduzido com métodos de software. Devido aos desalinhamentos causados pelo vôo em formação dos dois satélites a imagem obtida necessita de ser processada pixel por pixel, aumentando assim o tempo de execução do código.

Esta dissertação tem como objetivo estudar e implementar métodos que otimizem o código já desenvolvido, para que este tenha a capacidade de analisar mais imagens em um menor período de tempo.

O método de otimização que terá maior foco nesta dissertação é a paralelização. O código inicial será inicialmente paralelizado com OpenACC, sendo este um modelo de programação que permite que a paralelização do código seja feita apenas com o recurso a incrementação de diretivas. Com este modelo será estudado o desempenho da execução do código paralelizado para *CPU* e para *GPU* para poderem ser comparados ambos os desempenhos.

Devido à comunicação e transferência de memória entre o *CPU* e o *GPU* ser bastante demorada, a execução de código paralelo no *GPU* pode não ser eficaz se existir a necessidade de várias transferências de memória entre os dois processadores. Apesar de OpenACC permitir que o código seja paralelizado sem a alteração do mesmo, algumas partes do código para poderem ser paralelizadas necessitam de ser modificadas. Devido a ter partes do código a serem executadas no *CPU* e outras no *GPU* é necessário transferência de memória entre eles, fazendo assim com que o desempenho do código seja pior. Para obter a total paralelização do código de forma a eliminar o tempo gasto na transferência de dados o código foi alterado e codificado em *CUDA*. *CUDA* é uma *framework* da NVIDIA que permite escrever código para ser executado em *GPU*s da NVIDIA. Com o uso de *CUDA* é possível analisar a verdadeira potencialidade de um *GPU* na resolução de problemas matemáticos.

Palavras chave: PROBA-3, OpenACC, Paralelização, CPU, GPU, CUDA, Otimização

CONTEÚDO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE LISTAGENS

SIGLAS

A

ASPIICS Association of Spacecraft for Polarimetric and Imaging Investigation of the Corona of the Sun.

C

CPU Central Processing Unit.

CUDA Compute Unified Device Architecture.

D

DFT Discrete Fourier Transform.

DSM Distributed shared memory.

DSPS Digital Signal Processors.

E

ESA Agência Espacial Europeia.

F

FFT Fast Fourier Transform.

G

GPU Graphics Processing Units.

L

LASCO Large Angle and Spectrometric Coronagraph.

M

MCI Memory copy idiom, array assignment replaced by call.

O

OPENCL Open Computing Language.

S

SIMD Single Instruction, Multiple Data.

SMP Shared-memory multiprocessor.

T

TLP Thread-level Paralelism.

1

INTRODUÇÃO

ASPIICS é um novo coronógrafo solar externamente oculto que irá ser lançado a bordo da missão espacial PROBA-3 da *Agência Espacial Europeia (ESA)*. Nesta missão o ocultador externo será posicionado no primeiro satélite que está aproximadamente 150 metros em frente do segundo satélite, este irá conter o *optical instrument* (Llorente et al. , 2017) [15]. Durante 6 horas, em uma orbita de 19.38 horas, estes satélites irão orbitar em uma formação de máxima precisão com um erro entre eles de $\pm$ 15 mm longitudinalmente e $\pm$ 5 mm transversalmente, constituindo assim um coronógrafo externamente oculto de elevado tamanho.

Devido à grande distância entre o primeiro e segundo satélite será possível observar e estudar a coroa solar a altitudes extremamente baixas de aproximadamente 1,08 *solar radius*(R$\odot$), sendo que as altitudes entre 1,1 e 2,5 R$\odot$ ainda não tenham sido significativamente estudadas em observações da luz branca, apenas em *Large Angle and Spectrometric Coronagraph (LASCO)* (Brueckner et al. ,1995)[8] foi feito o estudo em altitudes entre 1,1 e 3,0 R$\odot$ tendo os resultados deste sofrido significativamente do fator da luz dispersa, nomeadamente ghost image.

Devido aos desalinhamentos causados pelo vôo em formação dos dois satélites não é obtida uma simples função de Green, como por exemplo em (C.Aime,2013)[16], logo sem uma deconvolução simples será necessário iterar pixel a pixel as imagens obtidas, tornando assim a missão em termos de software mais desafiante.

No estudo (Shestov et. al., 2019)[21] a fim de modelar e calcular todas as fontes de luz dispersa que iria entrar em *ASPIICS* foi utilizado o software Zemax Optic Studio, um software utilizado para o design e análise de sistemas de imagem e iluminação. Como o procedimento não pode ser implementado com uma convolução simples, existe a necessidade de utilizar uma linguagem de programação destinada mais à performance, como por exemplo: C, C++, Fortran ou Assembly.

Em (Shestov et. al., 2019)[21] foi utilizado Fortran em conjunto com OpenMP, apesar do uso desta linguagem (Fortran) os resultados não foram totalmente satisfatórios, tendo o procedimento total demorado 10 minutos a processar uma imagem de 4096 $\times$ 4096 pixeis em um computador com um processador de 40 Xeon e5-2580 núcleos, existindo assim a necessidade de otimização do código.

1.1 Enquadramento do projeto na missão espacial

Sendo um dos objetivos desta missão realizar um estudo à coroa solar, é necessário que para que este estudo obtenha bons resultados eliminar as influencias do ruído luminoso. Este ruído é em parte eliminado através de meios mecânicos más é também necessário realizar um processamento de imagem e uma eliminação de ruído através do software.

Em missões anteriores como por exemplo a missão LASCO, o coronógrafo era estático tornando assim o software bastante mais simples, onde era apenas necessário realizar uma convulsão de apenas uma iteração no software.

Nesta missão, como o coronógrafo será constituído por dois satélites a viajar a uma distância de 150 metros a velocidades consideráveis, estes irão gerar pequenos desalinhamentos no coronógrafo. Sendo que estes desalinhamentos não aconteciam em missões anteriores devido ao facto do coronógrafo ser estático. Será então necessário compensar estes desalinhamentos através de uma soma de milhares de iterações no software, fazendo assim com que este software se torne bastante mais desafiador.

Devido aos desalinhamentos existe então uma necessidade de se desenvolver um novo software para eliminar o ruído, sendo então este software desenvolvido pelo físico Sergei Shestov em (Shestov et. al., 2019)[21]. Apesar do sucesso do algoritmo na solução do novo problema que surgiu devido ao desalinhamento entre os dois satélites, este software é bastante lento fazendo assim com que o número de imagens obtidas pelos satélites sejam bastante menores do que o esperado.

Com o objetivo de possibilitar a obtenção de um maior número de imagens nesta missão será necessário realizar uma otimização ao código já desenvolvido, sendo este um dos objetivos desta dissertação.

1.2 Objetivos do projeto

O principal objetivo é estudar as vantagens e desvantagens da utilização de *Graphics Processing Units (GPU)* em comparação ao *Central Processing Unit (CPU)*, neste caso e em casos em que o problema é similar a este, e aplicar de forma adequada ao problema proposto.

1.3 Métodos de teste e de verificação

Os métodos de teste serão maioritariamente avaliados através da sua velocidade de execução e funcionalidade, sendo o desempenho do software o mais valorizado neste projeto.

Por fim, as verificações serão efetuadas comparando os outputs do código original com os outputs do código desenvolvido, comprovando assim que o algoritmo não foi corrompido e que este se

mantém com a mesma funcionalidade que o anterior. Será comparada também a velocidade de execução em ambos os códigos em diferentes setores dos mesmos.

1.4 Organização da dissertação

Esta dissertação está organizada em 10 capitúlos. Sendo que nos 3 capítulos iniciais será realizado um estudo mais teórico acerca do enquandramento do problema e do que foi feito até ao momento para tentar resolver este problema.

No capítulo 4 será comparado os resultados obtidos do código original com 2 compiladores diferentes. Em seguida nos capitulos seguintes irá ser realizada uma análise à possibilidade e aos beneficios da paralelização do algoritmo inicial.

Os capítulos da dissertação estão organizados de forma sequencial, ou seja, os resultados não serão apresentados apenas no final do documento, mas serão apresentados após cada iteração realizada na tentativa de otimizar o código.

2

ESTADO DA ARTE

Este capitulo é constituído por duas secções. Na primeira secção é analisada a evolução do estudo da coroa solar, sendo abordados os problemas que podem surgir durante o estudo da mesma e como ao longos dos anos estes problemas foram sendo resolvidos para que seja possível obter uma melhor compreensão acerca da coroa solar. No início da segunda secção foi feita uma breve explicação do porque da otimização do código que realiza o cálculo em *Association of Spacecraft for Polarimetric and Imaging Investigation of the Corona of the Sun (ASPIICS)* é necessária. Em seguida será discutida a performance dos GPU e de duas *frameworks* que permitem a sua programação. No final é feita uma explicação concisa sobre o desempenho computacional das *Fast Fourier Transform (FFT)*, e serão apresentadas técnicas já existentes para melhorar o seu desempenho.

2.1 Coronagraph

Apenas após 1930 se tornou possível estudar a coroa solar sem a existência de um eclipse solar. O pequeno período de tempo em que a a lua bloqueava totalmente os raios solares permitia aos cientistas estudar a coroa solar com um maior detalhe. Esperar para que um eclipse acontecesse já era uma limitação, sendo o pequeno período de tempo em que este acontecia uma outra grande limitação. Estas condições forçavam os cientistas a deslocar-se para diferentes partes do globo para efetuar estas observações e esperar que o eclipse solar acontecesse, com o risco de o céu estar nublado e a observação da coroa solar fosse impossível. Quando o céu estava limpo os cientistas tinham um pequeno período de tempo para recolher os seus dados e fazer as suas observações devido ao fenómeno, mas se não conseguissem recolher todos os dados ou realizar todas as observações necessárias a tempo estes teriam de aguardar vários meses para que surgisse uma nova oportunidade para realizar estes estudos.

Para ultrapassar estas limitações Lyot (Lyot et al. , 1933)[1] inventou o coronagraph. O coronagraph é um instrumento que permite o estudo da coroa solar sem a necessidade da existência de um eclipse solar. As limitações que este instrumento veio resolver foram as seguintes:

- **Necessidade de viajar** - Sem o coronagraph de Lyot era necessário viajar para pontos específicos do globo para estudar a coroa solar, a invenção do mesmo permitiu que a coroa solar pudesse ser estudada em qualquer parte do globo.

- **Espera por eclipses solares** - Com a invenção do coronagraph é possível estudar a coroa solar em qualquer dia do ano, logo que o céu esteja limpo.

- **Pressa para a obtenção de dados** - Com o coronagraph os cientistas têm controlo do tempo para obtenção dos dados da coroa solar.

Com as vantagens o coronagraph começou a ser utilizado por vários observatórios solares por todo mundo. O coronagraph de Lyot melhorou significativamente o estudo da coroa solar, mas este por si só não foi capaz de resolver todos os problemas que envolvem o estudo da coroa. Persistindo algumas limitações como, o céu tem de estar limpo para serem realizadas as observações, e mesmo quando este estava em perfeitas condições a parte mais exterior da coroa solar não era perfeitamente visível devido ao brilho existente no céu. Apenas na existência de um eclipse solar era possível visualizar a parte exterior da coroa solar, e mesmo quando este fenómeno acontecia o céu não é escuro o suficiente. Como consequências do céu não ser escuro suficiente as luzes dispersas influenciam negativamente os resultados.

Neste momento, com o uso de fotómetros polarizados os raios solares conseguiam ser seguidos até cerca de 3 raios solares($R\odot$) desde o centro do sol (Wlerick et al. 1957) [2].

Teoricamente era possível construir um coronagraph que não fosse afetado pela luz dispersa, e se este fosse posicionado em uma altitude maior do que a da dispersão de Raylight seria possível observar a coroa por completo. A primeira tentativa foi realizada por Newkirk and Eddy (NewKirk et al. , 1962)[3]. Estes utilizaram um balão de ar quente para levantar um *externally occulted coronagraph* a uma altitude de 24 quilómetros, mas esta tentativa falhou devido à elevada quantidade de luz dispersa originada na borda do ocultador.

Em outra tentativa, com um ocultador melhorado realizada em 28 de Junho de 1963, R.Tousey enviou um *externally occulted coronagraph* num foguete(Tousey, 1965)[4], permitindo assim observar mais além do que apenas com o coronagraph de Lyot. Neste lançamento for obtidas vinte e três fotografias de luz branca da coroa solar em uma altitude entre 131 e 204 quilómetros. Os resultados obtidos foram melhores que os anteriores, mas ainda com bastante margem para melhorar.

Desde então, vários *externally occulted coronagraph* têm sido lançados em satélites para observações a longo termo da coroa solar. Tais como: OSO-7 (1971-72) [5], Skylab (1973-74) [6], P78-1 (1979-1985) [5], the Solar Maximum Mission (SMM, 1980-1989) [7] and more. Cada uma destas missões demonstraram algumas melhorias no *coronagraph*, tais como:

- Melhor resolução espacial;

- Campo de visão;
- Tempo de resolução;
- Duração da missão.

Apesar de todas as melhorias, o *externally occulted coronagraph* tem duas limitações básicas. A primeira sendo que este instrumento consegue apenas fornecer imagens da coroa solar apenas por uma distância finita acima do limbo solar (>1.5 R⊙). A segunda limitação é correspondente à limitação de tamanho da abertura que não pode exceder alguns centímetros.

Para resolver estas duas limitações a missão LASCO (Brueckner et al. , 1995)[8], utilizou três *coronagraphs* diferentes com três campos de visão sobrepostos. Sendo, o primeiro de 1,1 até 3R⊙ da parte interior da coroa, no qual é utilizado uma versão do espelho igual à clássica do *coronagraph* de Lyot, mas sem o ocultador exterior. O segundo com um campo de visão de 1,5 até 6R⊙, devido à intensidade da luz a coroa solar diminui rapidamente comparativamente com a distância, sendo para estas distâncias utilizado um *externally occulted instrument*. O terceiro estende o campo de visão até 30R⊙. O segundo *coronagraph* sobrepõe os campos de visão do primeiro e do segundo de forma a obter resoluções espaciais de elevada dimensão durante todo o campo de visão sendo este de 1,1 até 30 R⊙. Esta missão teve um grande impacto na extensão do campo de visão da coroa solar, mas esta missão sofreu bastante com a presença de luz dispersa (Shestov et al., 2018)[9].

A adição de luz dispersa complica o diagnostico da temperatura do plasma e da intensidade dos eletrões no *coronagrap*(Wang et al.2017)[10], na imagem (Shearer et al. 2012; Goryaev et al. 2014)[11][12], no espectroscópico (Dolla δ Solomon 2008)[13] e até em observações de rádio (Högbom 1974)[14].

A nova missão da ESA (Llorente et al. , 2017) [15], com data esperada para ser lançada a meio do ano de 2021, será a primeira missão espacial a usar vôo de formação em precisão. Um par de satélites irão formar um *coronagraph* de 150 metros para estudar a coroa solar. Esta missão tem como objetivo a observação de luz branca na coroa solar começando por baixas altitudes $\sim$1.08R⊙, tornando esta a missão que obterá dados da coroa solar com altitudes mais baixas. Para obter uma boa performance na observação da luz branca a baixas altitudes é necessário reduzir os erros originados pela luz dispersa. A gama importante de altitudes desde 1.1R⊙ até 2.5R⊙ quase não foi estudada em relação a observações de luz branca realizadas no espaço, apenas estas observações foram realizadas na missão LASCO de altitudes de $\sim$ 1.1–3.0R⊙ nas quais as observações foram prejudicadas devido a luzes dispersas. *Coronagraph* clássicos não conseguem fornecer tais observações porque a difracção de luz por volta de 10^{-4} em 1.1R ⊙ excede o brilho teórico da coroa solar que é de cerca de 10^{-5} (Rougeout et al. 2017) [19].

No paper (Shestov et al. , 2018) [9] foi realizado o estudo e os cálculos dos desalinhamentos que irão ocorrer durante a missão Proba-3 que resultam da precisão do vôo em formação. Estes desalinhamentos irão afectar o desempenho do *coronagraph*, sendo que alguns destes são:

- O deslocamento relativo do sol ϕ= 10 arcsec relativamente ao deslocamento transversal dos satélites;
- A inclinação do *coronagraph* $\phi \sim$ 10 arcsec, devido ao deslocamento transversal dos satélites;
- A inclinação do satélite mais o desalinhamento do telescópio ϕ = 25 arcsec.

Devido a estes desalinhamentos é impossivel obter uma função simples de Green, tal como foi obtido por exemplo em (C.Aime,2013)[16]. Logo, como não é possível fazer apenas uma deconvolução simples é necessário iterar toda a imagem pixel por pixel, fazendo com que se torne um problema em termos de software mais desafiante.

De forma a calcular e modelar todas as fontes de luz dispersa que entrarão no aparelho foi utilizado o software Zemax Optic Studio [20]. Em (Shestov et. al., 2019) [21], a análise da *ghost light* foi baseada na abordagem de rastreamento de raios e levou em consideração efeitos tais como a dispersão nas lentes. Foi assumido que cada pixel da imagem de entrada representa uma onda de plano paralelo e foi calculada a resposta para cada componente do plano paralelo. Devido a este procedimento não poder ser implementado com uma convolução simples, o algoritmo deste não pode ser programado em linguagens de interpretação (tais como IDL, Python or MATLAB),ao invés deve ser programado em linguagens de compilação como Fortran. Logo, a linguagem utilizada em (Shestov et. al., 2019) [21] foi Fortran juntamente com OpenMP. OpenMP é um modelo de programação baseado em diretivas que tem como intuito melhorar o desempenho e portabilidade do código para diferentes processadores. Neste código o procedimento total demorava cerca de 10 minutos, em 100 iterações, com a entrada de uma imagem de dimensão de 4096 $\times$ 4096 num computador com um Xeon e5-2580. Sendo este tempo bastante demorado existe a necessidade de melhorar o desempenho do mesmo.

2.2 Otimização de código

A otimização do código é normalmente realizada após a finalização do estado de desenvolvimento, pois este pode reduzir a legibilidade e pode ser adicionado código para aumentar a performance do mesmo.

O código a ser estudado foi desenvolvido para realizar o cálculo da difracção em ASPIICS (Shestov et. al., 2019)[21] e a linguagem utilizada foi Fortran. O desafio computacional principal tem a ver com a necessidade de o ciclo principal execute entre 10.000 e 100.000 iterações, sendo que cada

iteração demora aproximadamente cinco segundos. Para 10.000 iterações com cada uma a demorar 5 segundos o tempo que o programa iria demorar a realizar o cálculo da difracção da luz branca em apenas uma imagem seria de aproximadamente 14 horas. Com recurso a técnicas de otimização e com a ajuda de um GPU o objetivo desta dissertação é de melhorar o desempenho deste programa.

2.2.1 *Graphics Processing Unit (GPU)*

O que torna a GPU uma ferramenta tão poderosa é a sua capacidade de resolver problemas complexos dividindo-os em milhões de pequenos problemas e trabalhá-los de uma vez só. A arquitetura do CPU é composta por um baixo número de núcleos com uma grande capacidade de memória cache mas que só pode lidar com poucas *threads* de cada vez. Ao contrário, a GPU é composta por milhares de "núcleos"que podem lidar com milhares de *threads* de cada vez, mas tem um menor espaço de memória.

CUDA *vs* OpenCL

Compute Unified Device Architecture (CUDA) é uma plataforma de computação em paralelo e um modelo de programação desenvolvido pela NVIDIA para GPUs [22]. CUDA permite aos seus utilizadores programar em múltiplas linguagens tais como : C, C++, Fortran, Python e Matlab. Os programas com CUDA consistem em duas partes. A primeira sendo a executada no CPU que é designado por *host*, e a segunda executada no GPU que é designado por *host*. Um dos compiladores da NVIDIA é encarregue de separar as duas partes durante a compilação.

Open Computing Language (OpenCL) [23] é uma *framework* desenvolvida pelo grupo Khronos. Esta *framework* foi desenvolvida para que fosse possível escrever programas que pudessem serem executados nas mais variadas plataformas, como CPUs, GPUs, *Digital Signal Processors (DSPs)* e outros tipos de processadores ou aceleradores de *hardware*. Ao contrário de CUDA, OpenCL apenas permite o uso das linguagens C e C++, e as plataformas como GPU não necessitam obrigatoriamente partilhar memória com o CPU (Paula, 2014)[24].

Enquanto CUDA apenas é suportado pela Nvidia, OpenCL é suportado pela AMD, NVIDIA, APPLE, INTEL e IBM. Devido a todos estes desenvolvedores terem a capacidade de gerir a portabilidade do código para todas as plataformas faz com que OpenCL se torne mais complexo do que CUDA. Porque OpenCL pode ser utilizado e programado para diferentes plataformas o código pode não ser otimizado para todos os dispositivos (Karimi et al., 2010)[25].

Tanto OpenCL como CUDA permitem programação em GPU e ambas invocam um pedaço de código que é executado num *kernel* do GPU. Alguns programadores afirmam que CUDA é mais eficiente(Karimi et al., 2010) [25], mas uma das vantagens de OpenCL é que permite a qualquer

criador de plataformas que implemente OpenCL no seu dispositivo, ao contrário do CUDA que apenas pode ser usado para produtos da NVIDIA. Alguns *papers* estudam o desempenho das duas *frameworks* tais como (Karimi et al., 2010) [25], e estes afirmam que apesar do acesso a várias plataformas OpenCL tem mais problemas de eficiência do que CUDA. No estudo realizado por Fang, Verbanescu e Sips (Fang et al. ,2011) [26], onde é comparado o desempenho de CUDA vs OpenCL, os resultados demonstram que na maior parte das aplicações CUDA obtém um desempenho 30% superior a OpenCL.

2.2.2 *Fourier Transform*

A *Discrete Fourier Transform (DFT)* de uma sequência complexa $x = x_0, ...x_{N-1}$ é uma sequência complexa de N pontos, $X = X_0, ...X_{N-1}$, onde:

$$X(k) = \sum_{j=0}^{N-1} x[n] e^{-jk\frac{2\pi}{N}n} \qquad k = 0, 1, ..., N-1 \tag{1}$$

A DFT inversa é definida de forma similar, sendo esta:

$$x[n] = \sum_{j=0}^{N-1} X(k) e^{jk\frac{2\pi}{N}n} \qquad n = 0, 1, ..., N-1 \tag{2}$$

FFT é um algoritmo criado para que a computação das DFTs tenha maior eficiência. Este algoritmo é bastante utilizado nas áreas da ciência e da engenharia, sendo este utilizado no cálculo da difracção em ASPIICS.

A FFT de uma sequência finita não periódica é nada mais do que uma DFT calculada através de um algoritmo especial que retira vantagem da simetria complexa conjugada e da periodicidade de *n* e *k* no exponencial complexo. Este algoritmo calcula a DFT em O(N log N) operações.

FFT With GPUs

A diferença principal entre computação da FFT no GPU e no CPU é que o valor de *n* é gerado como uma função do número da *thread t*, do bloco *b* e do número de *threads* por bloco *T*. Além disso, a iteração sobre os valores de *n* são gerados por múltiplas invocações no GPU ao invés de um ciclo único.

No estudo (Naga et al. , 2008)[27] em "High Performance Discrete Fourier Transforms on Graphics Processors"foi testado o desempenho da Transformada de Fourier em diferentes GPUs da NVIDIA e em um CPU com quatro núcleos. Neste estudo foram implementados e comparados algoritmos utilizando a biblioteca cuFFT da NVIDIA, uma implementação da Intel otimizada para o CPU e um algoritmo criado pelos investigadores utilizando CUDA.

O desempenho tanto no CPU como no GPU são melhores para *batched* FFTs comparativamente com FFTs únicas, logo durante o estudo todos os resultados obtidos foram com a utilização de *batched* FFTs. O desempenho do algoritmo desenvolvido por Naga K et al. numa FFT uni-dimensional é quatro vezes mais rápido do que com a biblioteca cuFFT e dezanove vezes mais rápido do que com CPU. Para uma FFT bi-dimensional esta é duas vezes mais rápida do que a biblioteca cuFFT e sessenta e uma vezes mais rápida do que com o CPU. Este estudo serve para comprovar que em condições normais a implementação de uma FFT no GPU tem normalmente um desempenho muito melhor comparativamente ao desempenho da mesmo no CPU

3

ANÁLISE DO CÓDIGO

Uma boa análise e estudo do código é essencial para uma otimização mais eficaz do mesmo. Neste estudo é necessário compreender as funcionalidades implementadas no código e quais os pontos em que é maioritariamente necessária a sua otimização. É fundamental detetar as secções mais demoradas deste algoritmo, sendo que a probabilidade de otimizar uma determinada secção ou função que tem um elevado tempo de execução é maior e mais benéfico do que otimizar uma que já tem um baixo tempo de execução.

3.1 Fortran

Para codificar o algoritmo foi utilizada a linguagem Fortran. Esta linguagem de programação é já bastante otimizada. Sendo esta desenvolvida com o propósito primário de resolver avaliações numéricas de formulas algébricas que necessitam de grande poder computacional ([29]). É devido aos recursos que dispõe que é bastante utilizada para fins científicos e militares, logo inicialmente a grande necessidade de poder computacional resultando que o algoritmo seja mais lento ou menos eficiente não será uma consequência do uso desta linguagem.

3.2 Estrutura do código

Na primeira análise foi estudada a constituição do código, ou seja a organização do mesmo. O algoritmo está representado por quatro secções como é possível observar na figura 1.

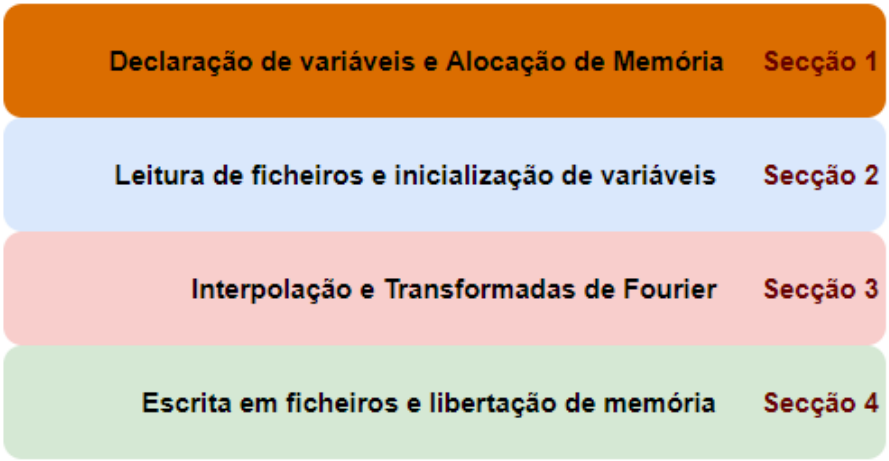


Figura 1: Secções Código

Começando por analisar a primeira secção é possível identificar alguns factores que indicam um possível motivo para que a execução do código seja um pouco mais demorada. Sendo estes, a declaração de vinte e uma matrizes de elevada dimensão com uma das seguintes características: complexas, de dupla precisão e reais, contendo cada uma destas perto de 17 milhões de dados. A elevada dimensão de cada matriz leva com que seja necessário um maior número de tempo para que estas sejam totalmente processadas nas secções seguintes.

Na segunda secção é onde está presente a leitura dos ficheiros necessários e onde os dados desses ficheiros são colocados nas suas variáveis correspondentes, é também a secção onde são configuradas e inicializadas as transformadas de Fourier e feitas algumas verificações e ajustes aos dados recebidos pelos ficheiros. Apesar de nesta secção já existir a manipulação e cálculo de matrizes de grande dimensão, não é a secção mais crítica em termos de performance em todo o algoritmo.

Na terceira secção sendo esta a mais complexa e a que merece mais atenção foi desenhada também uma figura para melhorar a sua compreensão e visualização (figura 2). Esta secção é constituída por um ciclo, onde o seu número de iterações originais é de 100. Este número ainda é relativamente baixo para que sejam alcançados os efeitos desejados do algoritmo. Este mesmo ciclo pode ser dividido em 23 pequenas subdivisões agrupadas em 3 divisões maiores como se pode observar na figura abaixo.

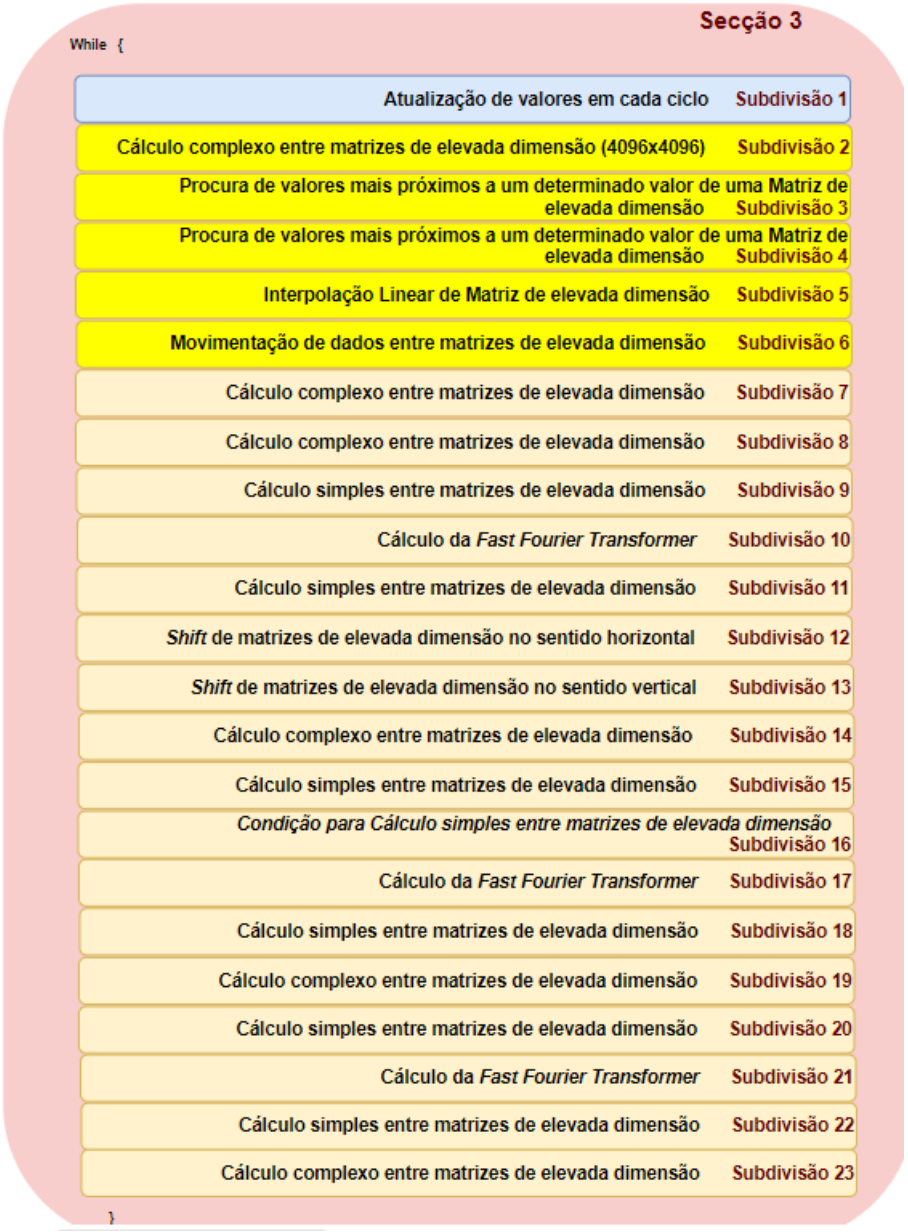


Figura 2: Subdivisões da Secção 3 do Código

A primeira divisão representada pela cor azul é constituída por apenas uma subdivisão que corresponde à atualização das variáveis a cada iteração, ou seja, esta subdivisão é utilizada para que estas variáveis sejam atualizadas para os respetivos valores destinados à iteração em questão, por exemplo, alguns destes valores estão guardados num *array* de 100 valores, sendo que cada um destes valores é destinado a cada uma das 100 diferentes iterações correspondentes. Esta divisão não apresenta características que representem um risco para a velocidade de computação do algoritmo, ou seja, não existe necessidade de otimização evidente.

A segunda divisão deste ciclo é constituída por 5 subdivisões sendo estas as responsáveis pela realização da interpolação dos dados. Em primeiro lugar são efetuados os cálculos necessários para a preparação da interpolação, em seguida, é feita a procura dos valores máximos e mínimos da matriz calculada anteriormente. Posto isto, é feita a própria interpolação, que analisando primariamente assume-se que deverá ser uma das subdivisões que requer mais poder computacional, por fim é feita a movimentação dos dados obtidos na interpolação. Na Fig.2 é observável que todas as operações efetuadas nesta divisão envolvem matrizes de dimensão elevada, contendo mais de um milhão e meio de dados, com os tipos: complexo, real e de dupla precisão, que necessitam de ser processados. Devido à enorme dimensão das matrizes processadas e da complexidade dos cálculos é necessário um elevado custo computacional.

A terceira divisão é constituída por 17 subdivisões, sendo esta a maior divisão das três. Esta divisão é constituída por:

- 5 subdivisões - Cálculos complexos
- 6 subdivisões - Cálculos simples
- 3 subdivisões - Cálculo da *Fast Fourier Transformer*
- 2 subdivisões - Shift de Matrizes de elevada dimensão
- 1 subdivisão - Em que o Cálculo simples só é realizado dependendo de uma condição

Analisando estes dados é possível observar que esta será a divisão que necessita maior poder computacional, não apenas pelo maior número de secções presentes nela, mas também pelo elevado número de cálculos, sendo estes complexos ou simples, a serem efetuados nestas matrizes. Estão também presentes três subdivisões onde são efetuados os cálculos das Transformadas de Fourier, sendo que estas também necessitaram de um elevado poder computacional. No seguimento da análise, esta divisão é a que requer maior atenção inicial.

3.3 Compilador

O compilador é um programa em que o seu objetivo é processar declarações escritas em uma determinada linguagem de programação e traduzi-las para uma outra linguagem, em que o processador da máquina que está a ser programada seja capaz de compreender. Para desempenhar as suas tarefas, o compilador deve executar dois tipos de atividades. A primeira atividade é a análise do código fonte, onde a estrutura e o significado do programa de alto nível são reconhecidos. A segunda atividade é a síntese do programa equivalente em linguagem simbólica, em geral estas duas atividades ocorrem praticamente em paralelo. Neste caso o compilador terá de traduzir o código

Fortran para linguagem de máquina quando executado. Para esse efeito foi então escolhido pelos programadores originais o compilador GNU FORTRAN (GFortran).

GFortran foi desenvolvido como parte do projeto GNU ([30]), este compilador é totalmente compatível com Fortran sendo assim uma opção bastante viável para a compilação do algoritmo desenvolvido no mesmo. GFortran permite também que na sua execução sejam invocadas *Flags* para a otimização do código. Sem a ativação de nenhuma opção de otimização o objetivo do compilador é reduzir o custo de compilação e fazer com que o *debugger* produza o resultado esperado. Se alguma *Flag* de otimização for ativa o compilador irá tentar melhorar a performance do código, tanto quanto ao seu tamanho ou ao tempo de compilação do mesmo. Esta ferramenta apresentada pelo compilador é bastante útil pois permite que o utilizador possa utilizar métodos de otimização no seu algoritmo sem um conhecimento profundo dos mesmos, fazendo assim com que o programador se possa concentrar mais na funcionalidade do código e também obter em simultâneo bons resultados de otimização.

3.4 Compilação e execução do código inicial

Como referido na secção anterior a linguagem de programação utilizada foi o Fortran com o compilador Gnu Fortran. Nesta secção inicialmente irão ser discutidas algumas *flags* de compilação utilizadas, em que o objetivo destas é indicar ao compilador a melhor forma de compilar o código. Em seguida, irão ser analisadas as otimizações realizadas pelo compilador e será feito um breve estudo acerca destas. Por fim, serão analisados e comparados os resultados de execução do código com e sem as otimizações efetuadas pelo compilador, para obter uma melhor compreensão sobre o efeito das otimizações nos tempos de execução do mesmo.

3.4.1 *Flags de compilação*

Para a compilação do código foram introduzidas as seguintes *flags* com os seguintes significados:

- -std=f2008 : Especifica o padrão expectável para o programa, neste caso Fortran 2008
- -ffree-form : Especifica a disposição utilizada no código. A *free form* foi introduzida a partir do Fortran 90 [31]
- -ffree-line-length-280 : Determina após qual coluna do código os caracteres devem ser ignorados. O valor padrão do número da coluna é de 132, mas neste caso foi definido o valor de 280
- -O3 : *Flag* de otimização, esta *flag* de otimização invoca todos os métodos de otimização disponíveis ao compilador GNU FORTRAN [32]

3.4.2 *Otimizações GNU FORTRAN*

Após compilar o código com as *flags* descritas acima, é possível visualizar onde foram efetuadas as otimizações e em que linhas estas foram efetuadas acrescentando uma outra diretiva, -fopt-info.

Na secção 3, descrita na Fig 2, os seguintes tipos de otimizações efetuadas pelo compilador foram:

- Subdivisão 2 : Vectorização do ciclo e *loop unrolling*
- Subdivisão 5 : Vectorização do ciclo , *loop unrolling* e vectorização básica de bloco
- Subdivisão 11 : Vectorização do ciclo , *loop unrolling* e vectorização básica de bloco
- Subdivisão 12 : Vectorização básica de bloco
- Subdivisão 13 : Vectorização básica de bloco
- Subdivisão 18 : Vectorização do ciclo , *loop unrolling* e vectorização básica de bloco
- Subdivisão 22 : Vectorização do ciclo , *loop unrolling* e vectorização básica de bloco

loop unrolling

Loop Unrolling é uma técnica utilizada para transformar o ciclo de modo a que o seu tempo de execução seja otimizado. Esta técnica basicamente consiste em remover ou reduzir iterações do ciclo. Com este método a velocidade de execução pode aumentar devido à eliminação do tempo do controlo da instrução de ciclo.

Um exemplo simples deste método pode ser o seguinte, o código deverá somar 16 valores de um *array* A com os do B e armazená-los no *array* c, este algoritmo pode ser codificado da seguinte forma:

```
1 for (int i=0; i<16; ++i)
      C[i] = A[i] + B[i];
```

Listing 3.1: Exemplo soma de arrays

Utilizando *Loop Unrolling* podemos obter algo como:

```
  for (int i=0; i< 16; i+=4) {
2     C[i]   = A[i]   + B[i];
      C[i+1] = A[i+1] + B[i+1];
4     C[i+2] = A[i+2] + B[i+2];
      C[i+3] = A[i+3] + B[i+3];
6 };
```

Listing 3.2: Loop Unrolling

Desenrolando assim o ciclo 4 vezes. Uma das grandes vantagens deste método é que se as instruções forem independentes podem ser executadas em paralelo, diminuindo assim o tempo de execução do programa. Uma das desvantagens é um aumento significativo do tamanho do código.

Vectorização do ciclo

Vectorização é o termo para o procedimento de converter uma instrução escalar numa vectorial. Grande parte dos processadores nos tempos correntes têm instruções do tipo *SIMD* ou *Vector*, estas instruções permitem ao processador efetuar 2 ou mais operações iguais simultaneamente.

Por exemplo, utilizando o código anterior :

```
for (int i=0; i<16; ++i)
    C[i] = A[i] + B[i];
```

Listing 3.3: Exemplo de soma de elementos de arrays

Onde C, A e B são *arrays* e pretende-se fazer o somatório de A com B e armazenar em C. Este somatório será feito 16 vezes individualmente e posteriormente armazenado o seu resultado no array C. Utilizando vectorização o processo é realizado de forma diferente, que pode ser igualado a algo como:

```
for (int i=0; i<16; i+=4)
    vectorizacao(&C[i], &A[i], &B[i]);
```

Listing 3.4: Exemplo de vetorização

A função "vectorizacao"irá utilizar recursos como instruções do tipo *SIMD* para realizar em vez de apenas 1 somatório por iteração (FIG. 3) 4 somatórios com apenas uma instrução (FIG. 4), cuja diferença para o método de *loop unrolling* é que este necessitava de 4 instruções para os 4 somatórios em cada iteração .

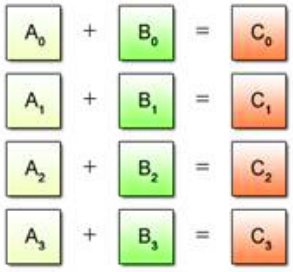


Figura 3: Operação Escalar

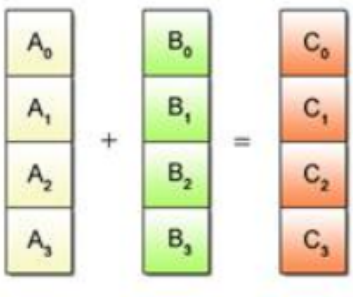


Figura 4: Operação SIMD

Alguns compiladores, como GFortran são capazes de computar vectorizações automáticas como esta, com recurso a *flags*. Para algoritmos mais complexos é necessária a ajuda do programador para gerar um código vectorizado ideal.

3.4.3 *Tempos de execução*

Nesta subsecção serão apresentados e analisados os tempos de execução de todas as subdivisões da secção 3 do código ilustrado pela Fig. 2.

GFortran Flags Ativas

No gráfico da Fig. 5 estão representados os tempos de execução de cada subdivisão ao longo das 100 iterações realizadas no ciclo da secção 3. Este gráfico é representa a tabela 1, em que esta utiliza o mesmo código de cores da figura 2. Neste ensaio foram utilizadas todas as *flags* descritas a cima com o compilador GFortran.

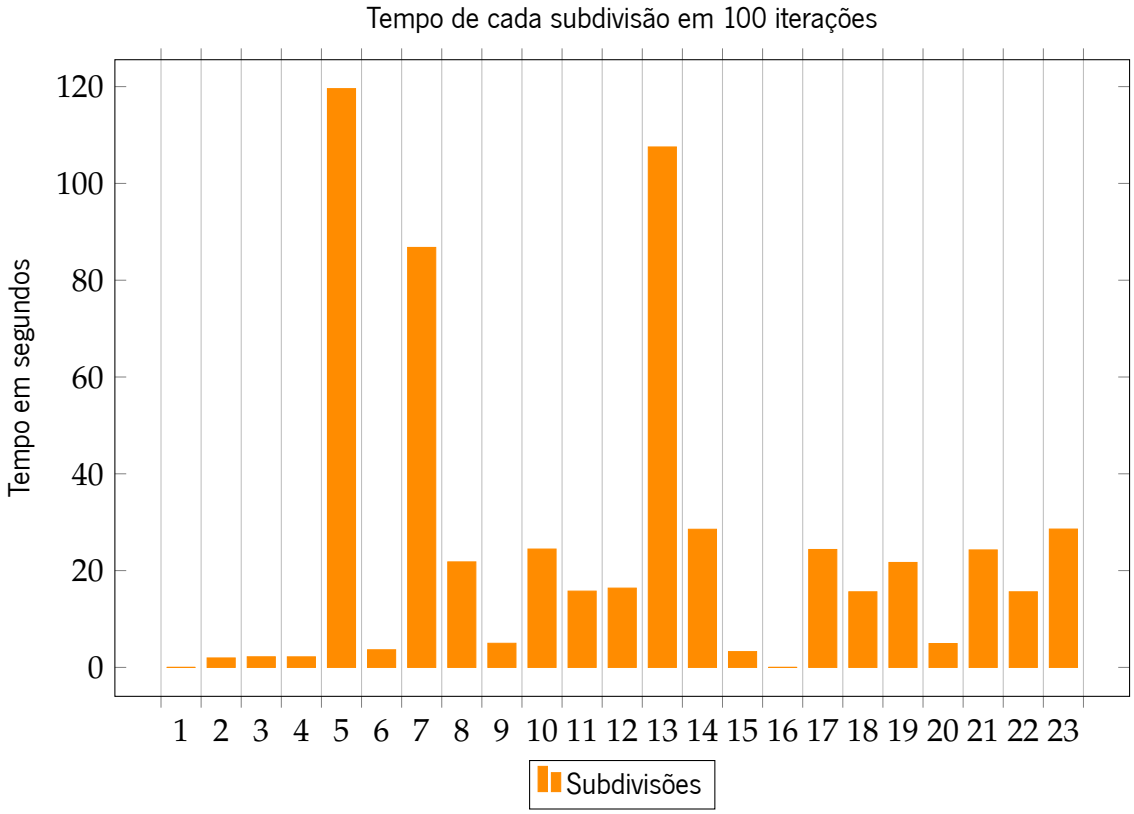


Figura 5: Gráfico de Compilação Com Flags GFortran

Tabela 1: Tempo de cada subdivisão em 100 iterações - Gfortran otimizado

Subdivisão	Tempo em segundos em 100 iterations	Subdivisão	Tempo em segundos em 100 iterations
1	0,00015	13	107,53
2	1,951	14	28,54
3	2,19	15	3,26
4	2,18	16	0,0001
5	119,58	17	24,36
6	3,64	18	15,62
7	86,77	19	21,68
8	21,8	20	4,93
9	4,98	21	24,27
10	24,45	22	15,62
11	15,76	23	28,58
12	16,38		

Resumindo, os valores deste ensaio são:

- Tempo total dos 100 ciclos : 574 segundos
- As subdivisões 5, 7 e 13 são as que necessitam de maior poder computacional, sendo que estas ocupam cerca de 55% do tempo de computação do ciclo
- Tempo total de cada ciclo em média : 5,74 segundos
- Tempo total da Divisão 1 : 129,5 segundos
- Tempo total da Divisão 2 : 444,5 segundos

Observando os dados obtidos é visível uma maior necessidade de otimizar as secções 5, 7 e 13, sendo que estas ocupam mais de metade do tempo de computação do código. A terceira divisão consome cerca de 3,4 vezes mais o poder de computação do que a segunda divisão, fazendo com que esta seja a que necessite de maior atenção.

GFortran Flags Desativas

No gráfico da Fig. 6 estão representados os tempos de execução de cada subdivisão ao longo das 100 iterações realizadas no ciclo da secção 3, este gráfico é representante da tabela 2 com o mesmo código de cores utilizados na Fig. 2. Neste ensaio para efeitos de comparação com o anterior, não foi utilizada a diretiva de otimização $-O3$.

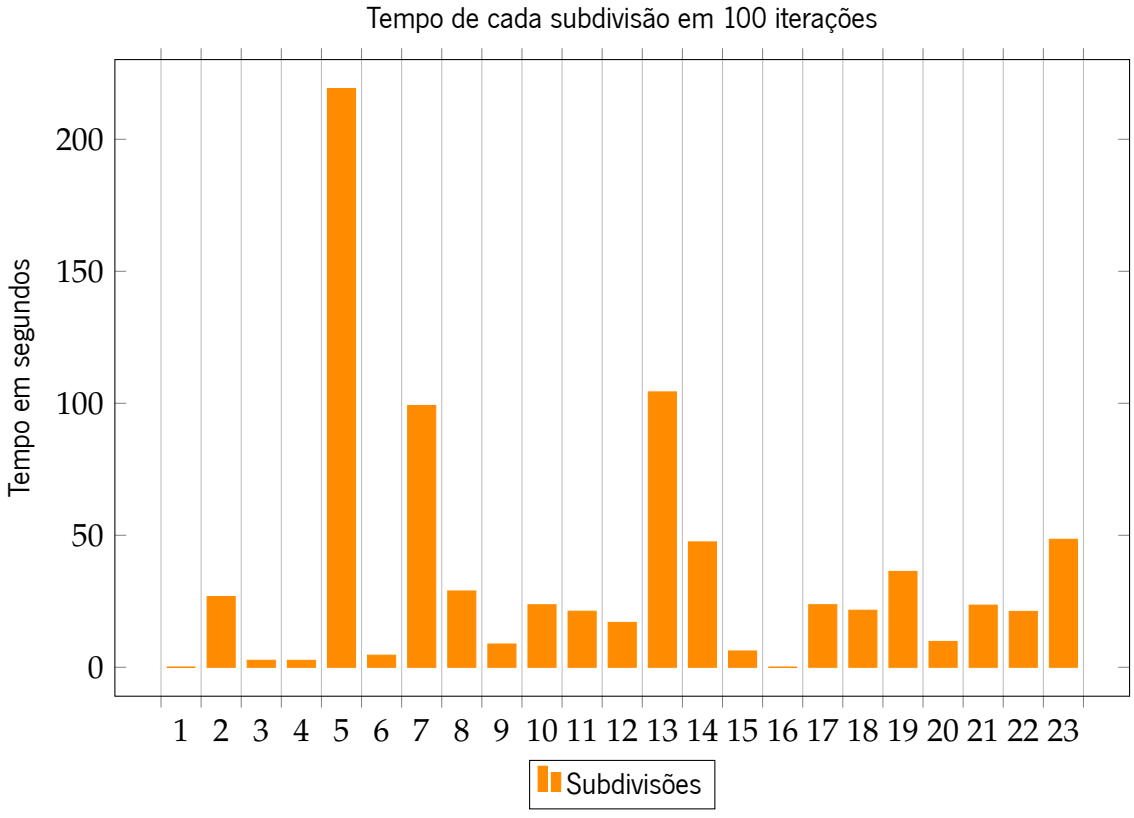


Figura 6: Gráfico de Compilação Sem Flags GFortran

Subdivisão	Tempo em segundos em 100 iterações	Subdivisão	Tempo em segundos em 100 iterações
1	0,00007	13	104,31
2	26,8	14	47,49
3	2,64	15	6,17
4	2,63	16	0,0001
5	219,23	17	23,7
6	4,58	18	21,6
7	99,16	19	36,3
8	28,9	20	9,78
9	8,81	21	23,54
10	23,68	22	24,54
11	21,23	23	25,54
12	16,99		

Tabela 2: Tempo de cada subdivisão em 100 iterações - Gfortran sem otimizações

Este ensaio demonstra varias informações bastante importantes tais como:

- Tempo total dos 100 ciclos : 797,18 segundos
- As subdivisões 5, 7 e 13 são as que exigem maior poder computacional, sendo que estas ocupam cerca de 53% do tempo de computação do ciclo
- Tempo total de cada ciclo em média : 7,97 segundos
- Tempo total da Divisão 1 : 255,88 segundos
- Tempo total da Divisão 2 : 541,30 segundos

Com os dados obtidos é visível novamente que existe a maior necessidade de otimizar as secções 5, 7 e 13, sendo que estas ocupam mais de metade do tempo de computação do código. Neste ensaio a terceira divisão consome cerca de 53% mais o tempo de computação do que a segunda divisão, fazendo com que esta seja a divisão que necessita maior atenção.

Comparação

Para uma melhor comparação dos diferentes desempenhos entre a compilação com e sem *flags*, os dados de cada subdivisão foram colocados lado a lado em um gráfico de barras presente na figura 7.

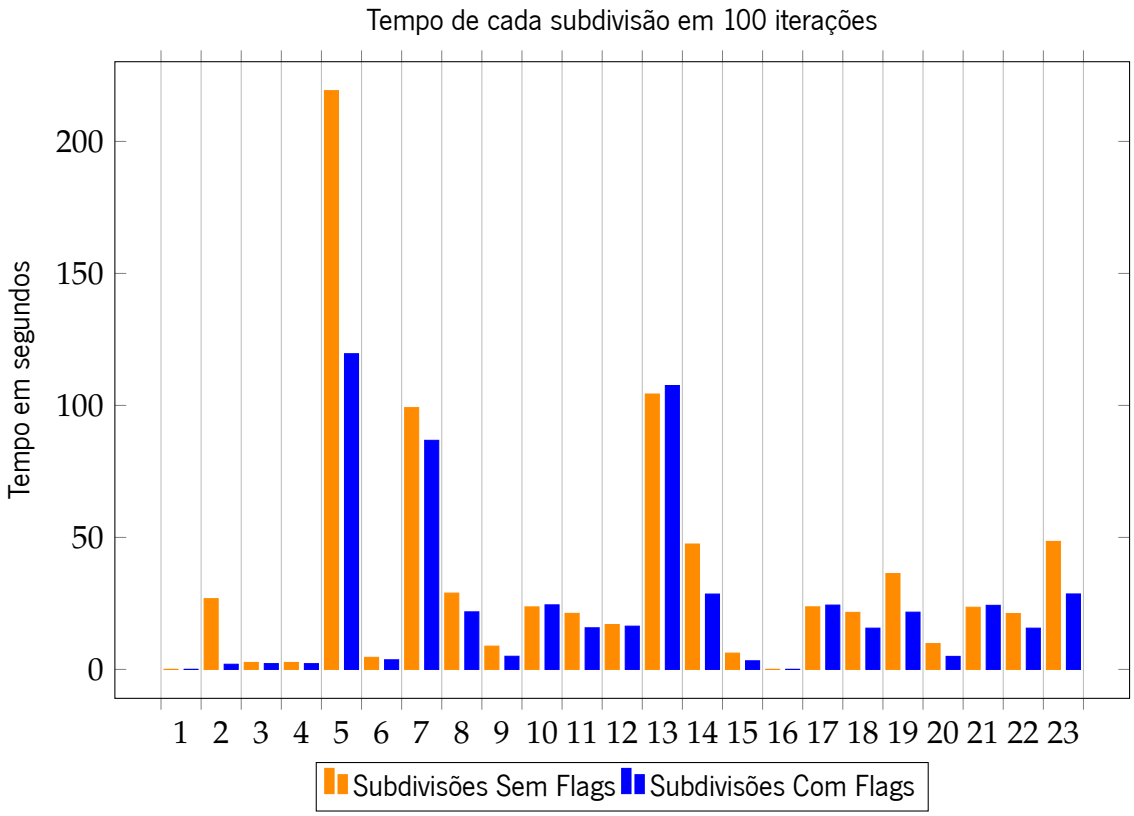


Figura 7: Gráfico de comparação - GFortran Com Flags vs Sem Flags

Comparando as duas execuções obtiveram-se os seguintes dados :

- A velocidade de execução da compilação com *flags* é de cerca de 30% mais rápida comparativamente com a compilação sem *flags*. Sendo esta 3 minutos e 43 segundos mais rápida;
- A subdivisão 2 com os métodos de otimização foi a que obteve maior aumento de desempenho, diminuindo o seu tempo de computação em 93%.
- A subdivisão 5 com os métodos de otimização é diminuiu o seu tempo de execução em 45%;
- As subdivisões otimizadas somam um total de 213,451 segundos com otimização e 374,5 sem otimização, diminuindo assim o seu tempo em 43%;

É possível concluir com os dados obtidos, que é possível aumentar bastante o desempenho do programa apenas accionando algumas *Flags* em compiladores como GFortran. Neste caso, o tempo de execução diminuiu em aproximadamente 18%.

4

COMPILAÇÃO E EXECUÇÃO COM PGI

4.1 Compilador PGI

O compilador da empresa PGI é o principal fornecedor de software e ferramentas para computação paralela, como por exemplo o OpenACC [34]. Este compilador tem suporte para *workstations*, servidores, e *clusters* que corram nos seguintes sistemas operativos: Linux, macOs e Windows. Como o objetivo é a utilização de OpenACC para o aumento do desempenho do código inicial o compilador a ser utilizado será o PGI.

Nesta secção irão ser analisados e demonstrados os resultados da execução e compilação do código original com o compilador PGI, e em seguida estudadas as otimizações realizadas por este compilador. Estes resultados serão depois comparados com os resultados obtidos na execução do mesmo código com o compilador GFortran com o uso de *flags*.

4.2 Compilação código inicial com PGI

Tal como GFortran, PGI tem também a opção de inserir *flags* de otimização para o compilador. A *flag* utilizada para a compilação correspondente à figura 8 foi *-fast*, esta tem como função ativar todas as *flags* de otimização disponíveis no compilador.

4.2.1 *Subdivisões otimizadas*

Com a *flag -fast* o compilador realizou as seguintes otimizações nas seguintes subdivisões:

- Subdivisão 2, 14, 19 e 23 : *Memory Copy Idiom*, *array assignment replaced by call to* pgf90_mcopy
- Subdivisão 3 e 4 : *Inlined*
- Subdivisão 5 : *Memory Copy Idiom*, *loop replaced by call to* __c_mcopy8
- Subdivisão 6, 9, 11, 12, 13, 15,16,18,20,22 : *Loop unrolled*

Inline Function

Inline é uma técnica de otimização utilizada em funções. Quando um programa executa uma instrução de chamada de função o CPU guarda o valor de memória do endereço da instrução posterior à chamada da função, em seguida copia os argumentos da função para a *stack* e só depois disto é que o controlo do programa é transferido para a função especifica. O CPU executa o código da função, guarda o valor que é retornado pela função no espaço de memória destinado e em seguida retorna para a localização posterior à chamada de função. Todo este processo pode se tornar uma sobrecarga, principalmente quando o tempo de execução da própria função é menor que o tempo da transferência entre a chamada da função e a execução da função em si. Este problema acontece normalmente em funções de pequena dimensão onde o tempo de execução é bastante pequeno, podendo se igualar ou até ser menor que o tempo da chamada da função. Para eliminar todo este tempo que não o tempo de execução da função em si, esta técnica consiste em substituir a linha correspondente à chamada da função pela função em si. Com o uso desta técnica é eliminada toda a sobrecarga adicional ao tempo de execução da função. Esta substituição é realizada pelo compilador durante o tempo de compilação, não pelo programador diretamente no código.

A recorrente utilização desta técnica irá aumentar o tamanho do código binário do ficheiro de execução, devido à duplicação do mesmo código. Poderá também aumentar o tempo de compilação se o código da função for alterado, pois o compilador irá ter que recompilar e substituir o código todo novamente. Um outro risco da utilização do *inlining* é a criação de lixo na memória do computador o que poderá posteriormente diminuir o seu desempenho.

Memory copy idiom, array assignment replaced by call

Com este método a otimização é feita da seguinte maneira, considerando o código seguinte em Fortran:

```
program exemplo

    integer(kind=4), dimension(4096) :: A, B

    A = 500
    B = A

end program exemplo
```

Listing 4.1: Memory copy idiom

Sem qualquer otimização as operações A = 500 e B = A irão sem realizadas com ciclos DO, mas com a otimização o compilador irá substituir o ciclo DO com a invocação da função mcopy, que é muito mais rápida. No caso do exemplo acima o ciclo DO iria ser substituido por mcopy4 porque se

trata de operações com inteiros, o uso de mcopy8 é para o caso (por exemplo) de variáveis do tipo *double*.

4.3 Execução código inicial com PGI

Os valores obtidos durante a execução do código inicial com as otimizações realizadas pelo compilador PGI, estão representados no gráfico da figura 8 para melhor visualização gráfica e na tabela 3 para melhor compreensão numérica. Os tempos de execução foram obtidos utilizando a função *cpu_time* que retorna o tempo de execução do CPU em segundos. Esta função é invocada antes e depois de cada subdivisão e é realizada a diferença enter os dois valores para obter o tempo de execução de cada subdivisão.

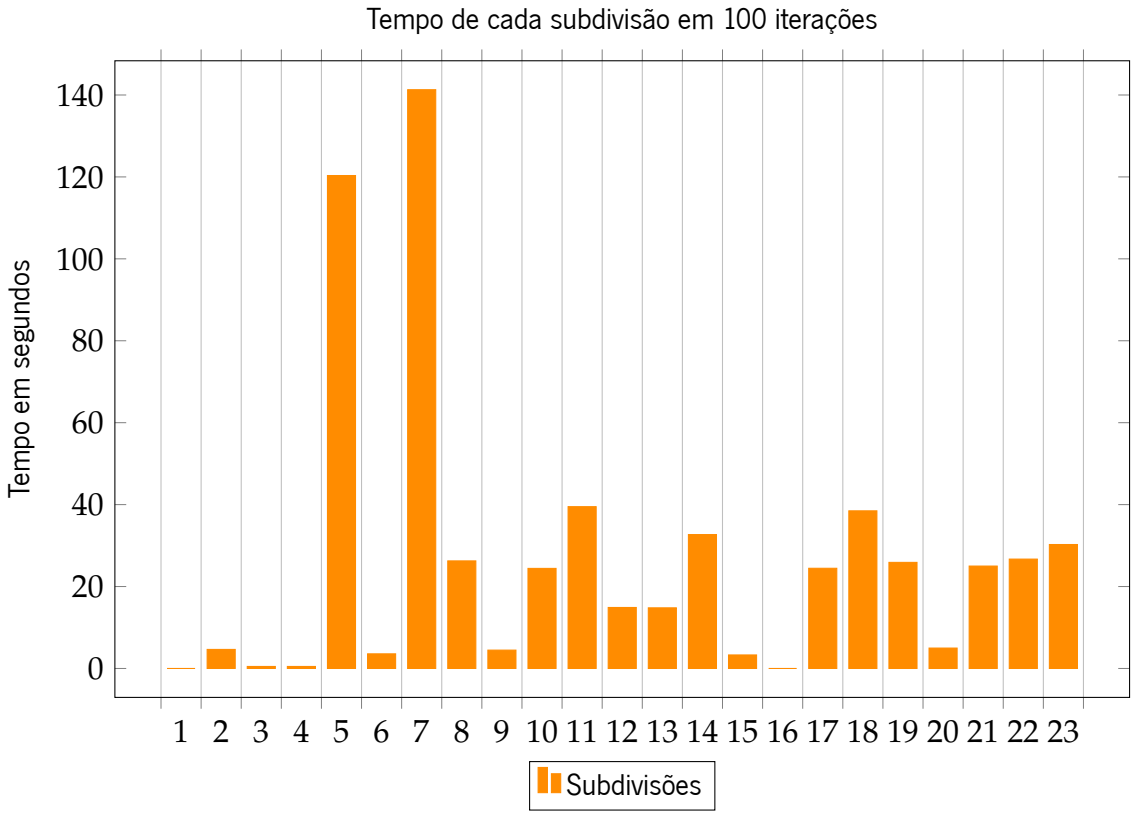


Figura 8: Gráfico de Compilação Com Flags - PGI

Tabela 3: Tempo de cada subdivisão em 100 iterações - PGI com otimizações

Subdivisão	Tempo em segundos em 100 iterações	Subdivisão	Tempo em segundos em 100 iterações
1	0,000001	13	14,8
2	4,65	14	32,96
3	0,49	15	3,3
4	0,49	16	0,000002
5	120,34	17	24,46
6	3,57	18	39,48
7	141,29	19	25,9
8	26,26	20	4,54
9	4,48	21	24,58
10	24,43	22	39,9
11	39,5	23	32,8
12	14,89		

Sintetizando, com este ensaio foram obtidos os seguintes resultados com maior relevância:

- Tempo total dos 100 ciclos : 623 segundos;
- As subdivisões 5 e 7 são as que exigem maior poder computacional, sendo que estas ocupam cerca de 42% do tempo de computação do ciclo;
- Tempo total de cada ciclo em média : 6,23 segundos;
- Tempo total da Divisão 2 : 129,54 segundos;
- Tempo total da Divisão 3 : 493,57 segundos;

4.4 Comparação PGI e GFortran

Os dados obtidos representados na tabela 3 e no gráfico da figura 8, referentes ao tempo de execução do código compilado utilizando PGI com a utilização de *flags* de otimização, e os dados representados na tabela 1 e no gráfico da figura 5, referentes ao tempo de execução do código compilado utilizando GNU Fortran com a utilização de *flags* de otimização, irão ser comparados para uma melhor compreensão das consequências de algumas das técnicas de otimização utilizadas por ambos.

Na tabela 4 estão representadas todas as subdivisões que foram otimizadas pelo compilador PGI e todas as otimizações realizadas pelo compilador GNU Fortran. Na quarta coluna desta tabela estão representados os valores da fracção dos tempos das subdivisões do compilador GNU pelas subdivisões do compilador PGI. Com este cálculo é possível analisar mais facilmente o impacto das otimizações efetuadas por cada compilador. As células da coluna 4 estão também identificadas por três cores, tendo estas os seguintes significados:

- **Branco -** Os valores do tempo de execução da subdivisão são semelhantes entre as duas compilações;
- **Vermelho -** Os valores do tempo de execução da subdivisão é mais rápido com o compilador GNU comparativamente ao compilador PGI;
- **Verde -** Os valores do tempo de execução da subdivisão é mais rápido com o compilador PGI comparativamente ao compilador GNU;

Tabela 4: GFortran vs PGI Fortran

Subdivisão	PGI	GNU	PGI	GNU
2	MCI	VC e Unrolled	4,65	1,951
3	Inlined	-	0,49	2,19
4	Inlined	-	0,49	2,18
5	MCI	VC, Unrolled e VCB	120,34	119,58
6	Unrolled	-	3,57	3,64
9	Unrolled	-	4,48	4,98
11	Unrolled	VC,Unrolled e VCB	39,5	15,76
12	Unrolled	VCB	14,89	16,38
13	Unrolled	VCB	14,8	107,53
14	MCI	-	32,69	28,54
15	Unrolled	-	3,3	3,26
16	Unrolled	-	0,000002	0,0001
18	Unrolled	VC,Unrolled e VCB	39,48	15,62
19	MCI	-	25,9	21,68
20	Unrolled	-	4,54	4,93
22	Unrolled	VC,Unrolled,VCB	39,9	15,62
23	MCI	-	32,8	28,58

Analisando a tabela temos que:

- A subdivisão 2 diminuiu para 42% do seu tempo original de execução com o uso dos métodos de vectorização e de *unrolling* do que com o método de *Memory Copy Idiom*, em que no decorrer de 100 iterações se traduz em aproximadamente 2,7 segundos
- O uso do método de *inline* torna as subdivisões 3 e 3 cerca de 3% mais eficazes
- A utilização do método de *Memory Copy Idiom* na subdivisão 5 apresenta o mesmo grau de eficácia do que o uso do método de vectorização, *unrolling* e vectorização básica de bloco
- A utilização da técnica de *unrolling* nas subdivisões 6 e 15 é ineficaz, e não apresenta otimizações no tempo de execução das subdivisões, este método faz também com que a velocidade de execução da subdivisão 12 e 13 seja maior do que com o uso da técnica de vectorização básica de bloco
- O método *unrolling* mostra-se eficaz na subdivisão 9,16 e 20 ao invés da não utilização de qualquer método
- A utilização do método de *Memory Copy Idiom* nas subdivisões 14, 19 e 23 faz com que o tempo de execução das mesmas seja mais demorado do que sem a utilização do método
- Nas subdivisões 11, 18 e 22 o método de *unrolling* é aproximadamente 64% mais eficaz em conjunto com os métodos de vectorização e vectorização básica de bloco do que isolado

Na Fig. 9 estão representados a laranja os tempos das subdivisões que correspondem à execução originada pela compilação do compilador PGI, e a azul as subdivisões que correspondem ao compilador GNU. Na totalidade os tempos de execução das 100 iterações com o PGI foram de 623 segundos e as com GNU de 574 segundos. Tendo estas comparativamente uma diferença de 49 segundos na totalidade ou de 490 milissegundos por iteração. As maiores diferenças entre os dois compiladores são nas divisões, 7 com uma diferença de 550 milissegundos mais rápida por iteração com o uso de GNU Fortran, 11 com uma diferença 240 milissegundos mais rápida por iteração com o uso de GNU Fortran, 13 sendo a execução referente a PGI 930 milissegundos por iteração mais rápida do que com o uso do compilador GNU, 18 com uma diferença de 238 milissegundos por iteração sendo o GNU mais rápido e a subdivisão 22 sendo a execução referente à compilação com o compilador PGI mais lenta 243 milissegundos por iteração. Algumas destas diferenças são explicáveis com o uso de diferentes métodos de otimização, que otimizam as subdivisões fazendo assim com que estas sejam mais rápidas do que as mesmas utilizando com outros métodos, como é observável na tabela 4. A subdivisão 7 apresenta uma diferença significativa entre os dois compiladores, apesar de que nenhum destes apresenta algum tipo de mensagem sobre a otimização da mesma através das *flags* correspondentes para o efeito.

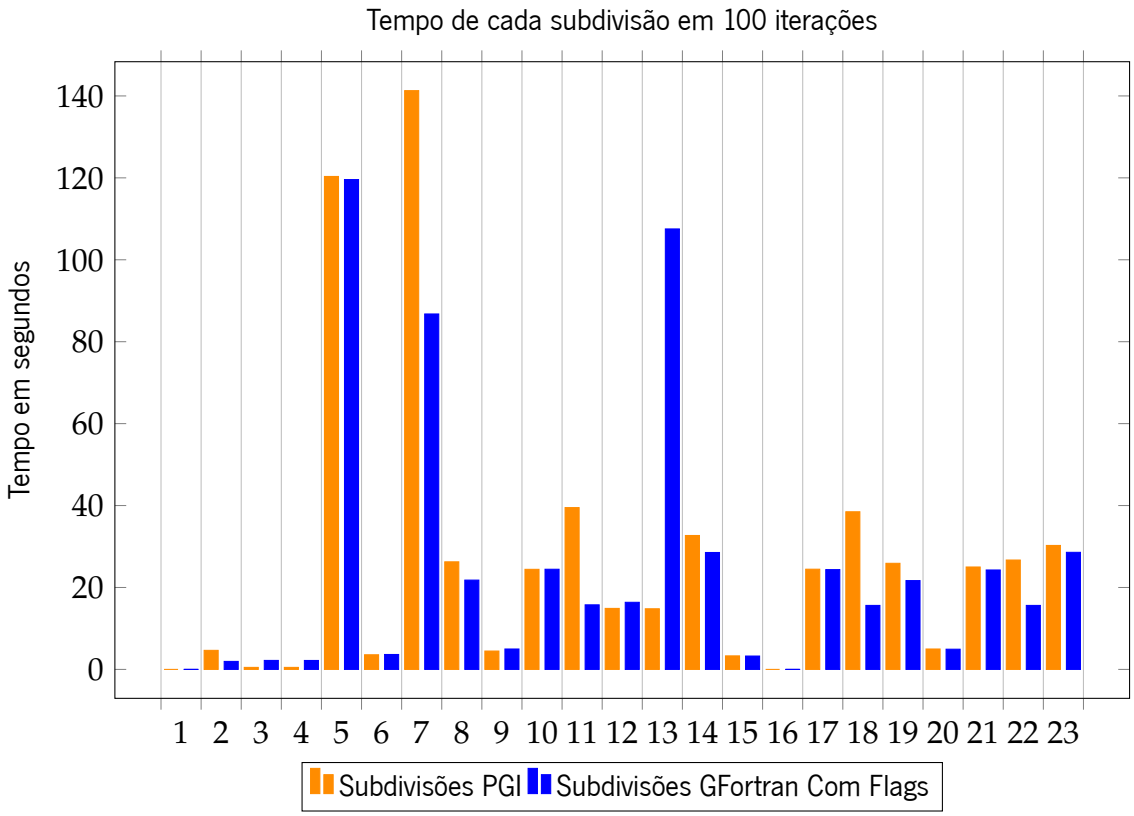


Figura 9: Gráfico de comparação - PGI vs GFortran

5

TÉCNICAS DE OTIMIZAÇÃO DE CÓDIGO

Técnicas para a otimização de código são cada vez mais importantes devido ao alto desempenho esperado em todas as aplicações utilizadas nos dias de hoje. Milissegundos podem fazer a diferença entre, por exemplo, um automóvel evitar uma colisão com um objeto e salvar uma vida, ou de uma câmara não obter a imagem perfeita de um veículo que não pagou a portagem. A velocidade de computação é essencial nestes casos e em muitos outros, dai a necessidade de otimizar e tirar o maior proveito do algoritmo criado.

Algumas destas técnicas de otimização sendo *loop unrolling*, vectorização do ciclo e vectorização básica do ciclo, já foram analisadas na secção 3.4 e outras como *inlining* e MCI foram analisadas na secção 4.2.

5.1 As três formas de paralelizar

Atualmente, a grande parte dos processadores presentes no mercado tem capacidade de processamento em paralelo, como por exemplo, os telemóveis com 2/4/8 *núcleos*, computadores que com o uso de *workstation* tem a capacidade de chegar de 32 a 64 *núcleos* e os GPU que tem milhares de *threads*. Logo, com todos estes recursos, desenvolver código em série não é mais a forma mais rápida ou eficiente para obter o melhor desempenho nas aplicações.

Até aproximadamente 2005, o desenvolvimento de processadores era focado maioritariamente em aumentar a sua frequência de processamento e a sua performance em uma única *thread* [33], mantendo-se assim o número de *núcleos* em 1 até então. Desde 2005 é observável uma estagnação dessas duas características e um aumento exponencial do número de *núcleos* lógicos. No futuro prevê-se que o número de *núcleos* irá continuar a aumentar de forma exponencial, já pelo contrário a frequência e a performance de um só *core* prevê-se que não terá grande crescimento no futuro. Com estes dados, é perceptível a grande importância da aprendizagem da programação paralela pois espera-se que o futuro do processamento seja maioritariamente baseado em paralelismo, ao invés, de processamento em série.

A paralelização pode ser realizada de três diferentes formas (Fig. 10), por exemplo, quando existir a necessidade de utilizar álgebra linear, transformadas de Fourier, gerar números aleatórios ou realizar multiplicação de matrizes, a forma mais simples de retirar vantagens do paralelismo é através de bibliotecas. Normalmente todos os vendedores têm a sua própria biblioteca para o uso de funções matemáticas, e a probabilidade de essas bibliotecas poderem ser otimizadas é muito baixa.

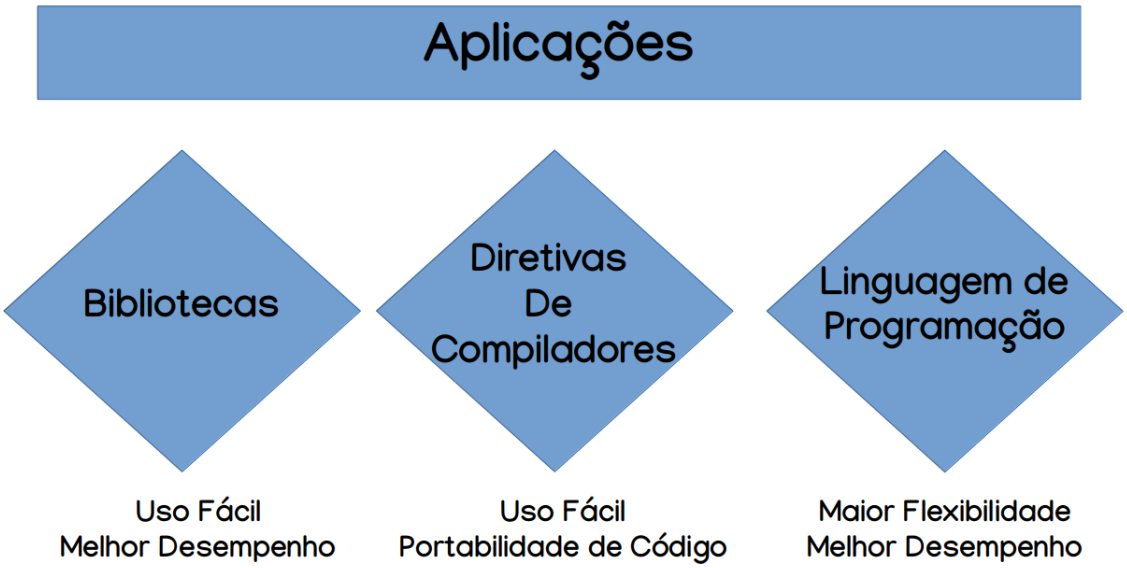


Figura 10: Três formas de otimizar

Outra forma de otimização é através do uso de diretivas do compilador. A utilização de diretivas é uma das formas utilizadas em que o utilizador normalmente se sente mais confortável. Este pode programar com a linguagem que se sente mais confortável e passar informação adicional ao compilador, para que este possa fazer a paralelização do código por si. Logo qualquer algoritmo pode ser escrito na maior variedade de linguagens e mesmo assim muito provavelmente o ”trabalho pesado”ser realizado pelo compilador ao invés do programador.

A terceira e última forma de paralelizar uma aplicação é com a utilização de linguagens mais orientadas a esse propósito, como por exemplo CUDA ou OpenCL que são mais orientadas a programação paralelizada. Estas linguagens vão permitir ao programador um melhor controlo do dispositivo a ser programado e fazer com que a aplicação demore um menor tempo de execução na máquina pretendida. Se o objetivo for que esta aplicação tenha um menor tempo de execução em dois tipos de máquinas diferentes são então necessárias duas versões da mesma aplicação, sendo uma para cada uma das máquinas. Sendo assim a curva de aprendizagem maior com o uso deste tipo de linguagens, e a carga de trabalho é maior não só na codificação como na manutenção do algoritmo.

5.2 Análise do código - paralelização

A viabilidade de pegar num código sequencial e desmembrá-lo para o executar em várias peças em paralelo depende se estas peças necessitam umas das outras para a sua realização. Para que o código tenha um bom desempenho paralelo é necessário que este respeite a lei de Amdahl[36]. Esta lei diz que, a quantidade de desempenho a ser ganha pelo uso da computação em paralelo é eficazmente limitada pelas peças do programa que tem de ser executas em ordem. Para *n* processadores, a única forma do programa executar *n* vezes mais rápido é se este conseguir ser dividido em *n* partes independentes do mesmo tamanho, sendo isto raramente possível.

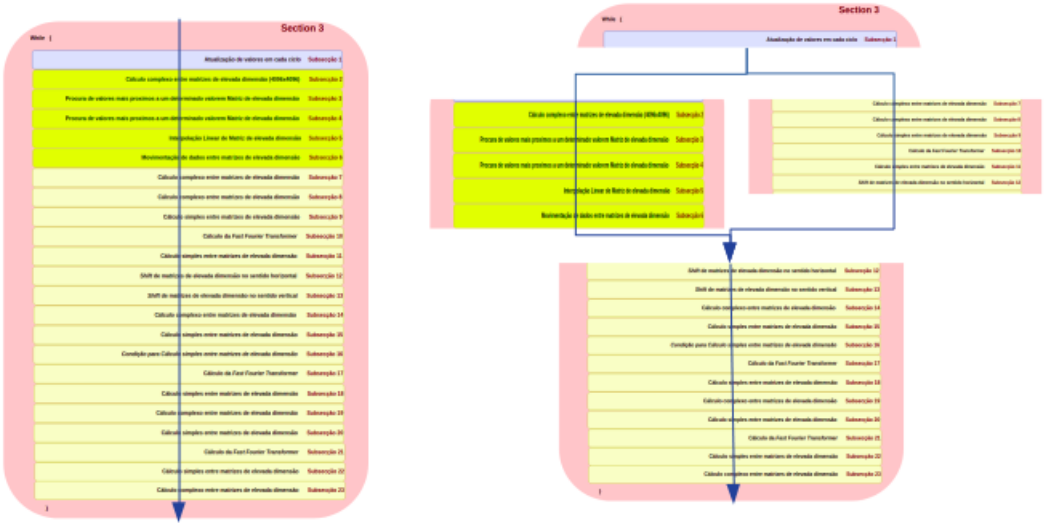


Figura 11: Paralelização desigual do código

Se um cálculo utilizar o resultado do cálculo anterior é necessário esperar que o primeiro cálculo seja realizado para realizar o cálculo seguinte, isto é considerado uma dependência computacional porque influencia o fluxo computacional do programa. Ou seja, seria possível realizar a paralelização representada na figura 11 se a subdivisão 7 não for dependente da subdivisão 8. Apesar do desempenho não ser melhorado por um fator de 2 devido às divisões não serem iguais iria observar-se uma diminuição do tempo de execução do programa. Mas todas as subdivisões, representadas na figura 2, são dependentes da subdivisão seguinte, logo a execução de duas ou mais subdivisões em paralelo torna-se impraticável.

Além de não ser possível paralelizar a secção 3 da forma representada na figura 11, ou seja cada *thread* executar cada subdivisão individualmente devido à dependência de todas as subdivisões, também não é possível paralelizar o código da forma representada na figura 12. Na figura 12 a forma de paralelização consiste em processar duas ou mais iterações ao mesmo tempo. Mas todas as iterações do ciclo da secção 3 são dependentes da iteração anterior, significando que são

necessários os dados da iteração $i - 1$ para a iteração i, tornando assim as iterações sequências e dependentes umas das outras.

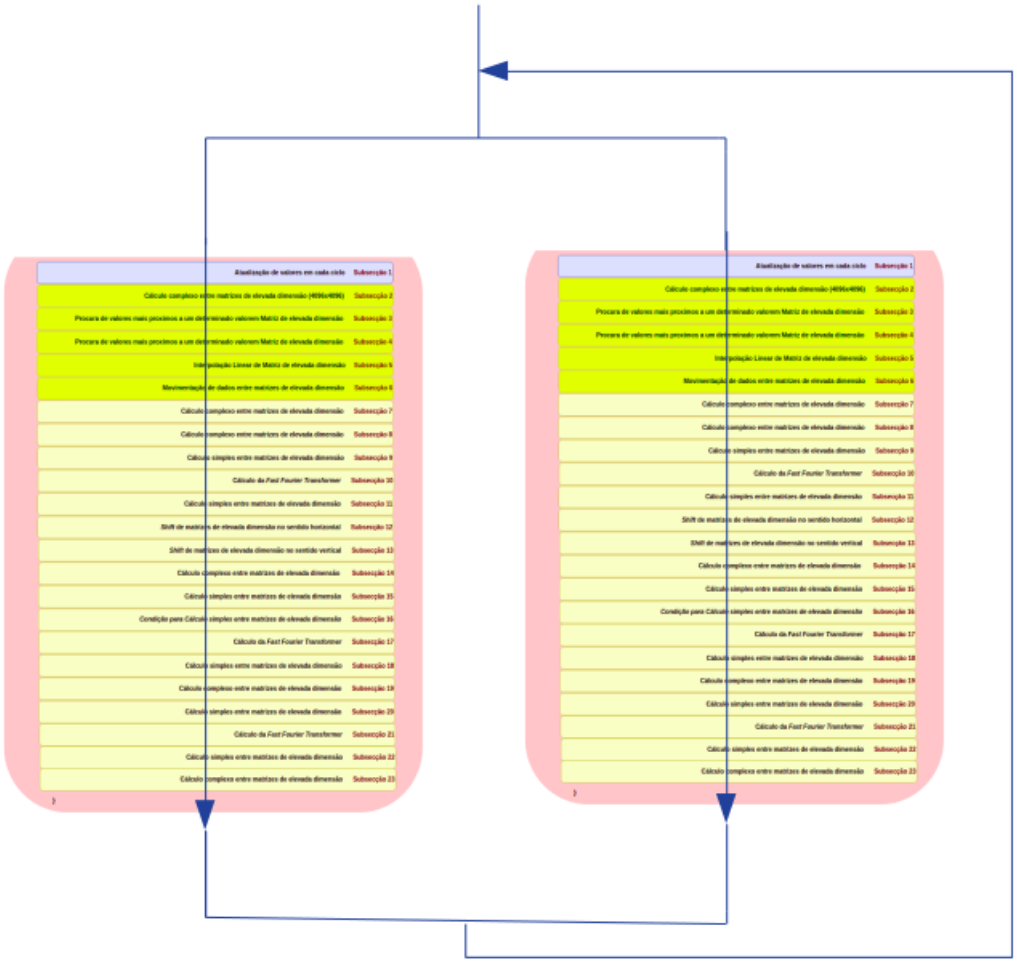


Figura 12: Paralelização por iteração

A maioria das subdivisões da secção 3 são operações efetuadas em matrizes de elevada dimensão. Apesar de estas operações com a linguagem Fortran poderem ter a sintaxe do tipo:

```
1
C = A + B
```

Listing 5.1: Operação escalar

Devido a serem operações com matrizes de tamanho 4096 por 4096 podem também serem escritas como:

```
1
do i= 1 , 4096
3     do n = 1, 4096
          C(i) = A(i) + B(i)
```

```
5       end do
    end do
```

Listing 5.2: Operações em matrizes de dimensões 4096x4096

Com a estrutura utilizada no último exemplo de código é facilmente observável a existência de um ciclo de elevado tamanho. As subdivisões, 2, 7, 8, 9, 11, 14, 15, 18, 19, 20, 22 e 23 podem ser representadas com uma estrutura similar à deste exemplo, apesar de teren cálculos com maior complexidade. Nenhuma destas subdivisões apresenta dependências nas operações efetuadas nos seus ciclos respetivos, logo existe a possibilidade de paralelização. As paralelizações efetuadas terão uma estrutura similar à da figura 13, em que esta representa como a paralelização da subdivisão 2 seria feita com o recurso de 4 *threads*.

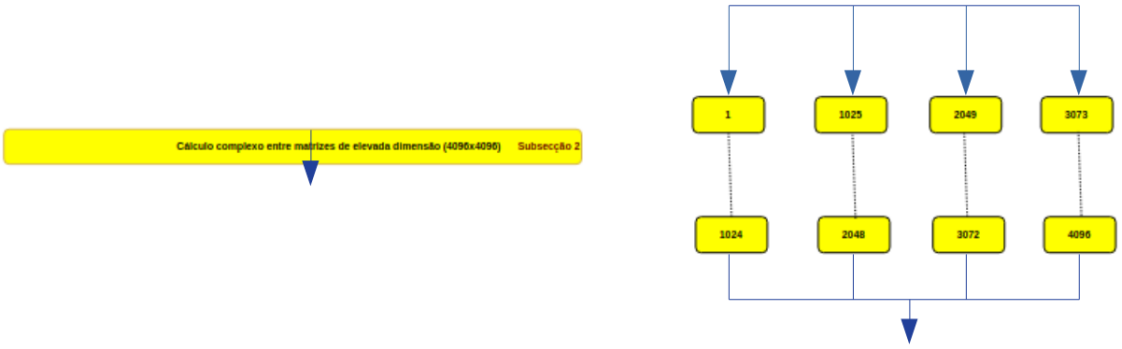


Figura 13: Paralelização por subdivisão

5.3 OPENACC

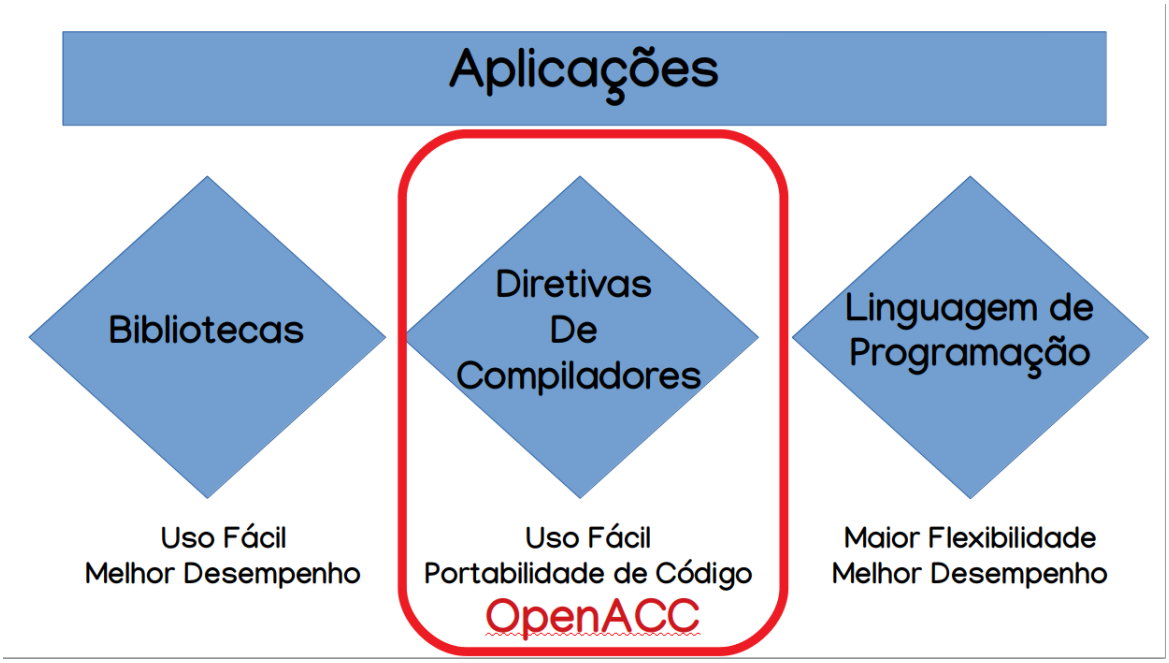


Figura 14: Três formas de otimizar - OpenACC

5.3.1 *O que é o OpenACC ?*

OpenACC é um modelo de programação em paralelo baseado em diretivas, desenhado com o intuito de desempenho e portabilidade. Desenvolvido por Cray, CAPS, Nvidia e PGI, sendo este modelo de programação suportado pelo compilador PGI. Este modelo de programação permite que o código seja mais rápido, tenha maior performance e uma grande portabilidade para diferentes plataformas. Ou seja, este modelo permite que o programador não precise de ter em atenção para que tipo de processador o código esta a ser desenhado, por exemplo com o uso de OpenACC é possível que o mesmo código seja executado em paralelo num GPU ou em paralelo num CPU ou mesmo em série em um CPU, tudo isto apenas com o uso de diretivas e sem a necessidade de reescrever o código. Este modelo de programação é bastante útil para estudar se existe vantagens entre paralelizar uma função ou executá-la em série, e se for vantajoso paralelizá-la se a sua performance é melhor em num CPU multicore ou em um GPU.

Com OpenACC é possível utilizar diretivas do compilador para indicar ao mesmo como compilar o código da forma mais eficiente, em C/C++ estas diretivas têm o nome de *pragmas* e em Fortran elas são conhecidas como diretivas e têm um formato de comentário especial. OpenACC fornece suporte sobre como criar execuções em paralelo em qualquer plataforma, como GPU e CPU onde os dois tem diferentes espaços de memória sendo que, OpenACC tem formas de gerir as memória entre as duas plataformas. É de notar que isto é tudo realizado apenas com a incrementação de linhas de código sem a necessidade de alteração do código inicial.

Resumindo OpenACC é :

- **Incremental:** O código inicial não necessita de ser alterado, é possível simplesmente incrementar linhas de código ao código original e mesmo assim continuar a otimizá-lo
- **Single Source:** Pragmas ou diretivas no caso de fortran, podem sem implementadas em qualquer plataforma sem qualquer alteração do código. O compilador determina como paralelizar para as diferentes plataformas
- **CPU, GPU, Vários núcleos:** OpenACC corre em vários GPU ou CPU e tem a capacidade de gerir a memória entre todas as plataformas
- **Interoperável :** É possivel o programador utilizar funções da Nvidia com OpenACC
- **Desempenho e Portabilidade:** OpenACC tem um elevado desempenho em qualquer processador e pode ser facilmente reutilizado numa plataforma diferente
- **Pequena curva de Aprendizagem:** Este modelo foi projetado para ser facilmente utilizado e entendido, pode ser programado em diferentes linguagens como C, C++ e Fortran. O programador não necessita de saber detalhes de baixo nível do *hardware* para poder utilizá-lo de forma eficiente

O compilador pode também ignorar algumas adições de OpenACC feitas no código, logo o mesmo código pode ser utilizado de forma paralela ou sequencial.

Este modelo de programação tem suporte para as seguintes plataformas:

- **POWER**
- **Sunway**
- **x86 CPU**
- **AMD GPU**
- **NVIDIA GPU**
- **PEZY-SC**

5.3.2 *Sintaxe OpenACC*

Figura 15: Sintaxe OpenACC

Em C/C++ existe o conceito de pragma. Pragma é uma instrução para o compilador que é externa ao C e ao C++, por outras palavras, pragmas não fazem parte de C ou C++. No caso de o compilador não entender um pragma em específico pode simplesmente ignorá-lo.

Em Fortran a diretiva é formatada da mesma forma que um comentário, que tal como em C/C++ dá instruções ao compilador sobre como compilar o código.

A palavra "*acc*"informa o compilador que será utilizada uma diretiva de OpenACC.

As *clauses* são especificadores ou adições às diretivas.

No código seguinte está representado um exemplo de aplicação de diretivas num código em Fortran, onde a primeira diretiva é relacionada com a gestão de dados e a sua movimentação, a segunda diretiva informa a iniciação de uma possível execução em paralelo, a terceira e última indica uma possível otimização do ciclo e do seu mapeamento. Todas estas diretivas serão estudadas em secções posteriores.

```
!$acc data copyin(a,b) copyout(c)
2
    ...
4   !$acc parallel loop

6       !$acc loop
        do i = 1 , 16
8           C(i)= A(i) + B(i);
            ...
10  ...
```

Listing 5.3: Estrutura diretivas OpenACC

5.3.3 *OpenACC Diretivas Paralelas*

Depois de identificadas as partes do código que podem ser otimizadas sem qualquer alteração do código original, para que este se possa tornar paralelo ao invés de sequencial com o recurso do modelo OpenACC, será necessária a incrementação de diretivas ao código. Estruturalmente exemplificadas na figura 15, para que o compilador consiga realizar a paralelização da parte do código indicada por essas diretivas, se possível.

Diretiva Kernels

A diretiva *kernels* identifica a região do código da qual existe a probabilidade de esta ser paralelizada, mas com o uso desta diretiva o programador depende das capacidades de paralelização do próprio compilador para analisar a região, identificar os ciclos que deverá paralelizar e acelerar os mesmos. O código abaixo demonstra o uso desta diretiva com a linguagem Fortran.[35]

```
!$acc kernels
do i=1,N
    y(i) = 500
    x(i) = i
end do

do i=1, N
    y(i) = 2.0 * x(i) / y(i)
end do

!$acc end kernels
```

Listing 5.4: Estrutura diretiva kernels

Neste exemplo são inicializados dois *arrays* e depois é efetuado um cálculo simples com eles. De notar que foi indentado um bloco de código com a utilização de duas diretivas sendo a inicial "!$acc kernels"e a final, "!$acc end kernels". Com estas duas diretivas o compilador irá identificar este bloco de código e em seguida analisar se algum dos ciclos tem dependência de dados. Se todos os dados forem independentes o compilador irá paralelizar os dois ciclos gerando um *kernel* para cada um deles. Neste caso o compilador tem total liberdade para determinar como mapear da melhor forma a paralelização disponível nestes ciclos para o hardware em que serão executados. Um cuidado a ter com esta diretiva é que se o compilador não tiver a certeza de que os dados do ciclo são independentes o ciclo não será paralelizado.

Diretiva Parallel

A diretiva *parallel* é utilizada para identificar uma região do código que será paralelizada com OpenACC *gangs*. *Gangs* é um termo utilizado que pode significar diferentes coisas dependendo do processador a ser utilizado. Num CPU multicore, na generalidade *gangs* é igual a *thread* e num GPU, *gangs* equivale a um *thread block*. O código seguinte mostra a possível utilização desta diretiva.

```
!$acc parallel
do i=1,N
    y(i) = 500
    x(i) = i
end do

!$acc parallel
do i=1, N
    y(i) = 2.0 * x(i) / y(i)
end do
```

Listing 5.5: Estrutura diretiva parallel

Com o uso da diretiva *parallel*, OpenACC irá gerar 1 ou mais *gangs* em paralelo, sendo estes *gangs* por exemplo CPU *threads*. Em seguida irá atribuir a cada um deles as instruções do código localizadas após a diretiva. Com recurso a quatro *threads* de um CPU no exemplo acima, todos elas iriam realizar exatamente as mesmas funções, tornando assim o código redundante. Ou seja, cada *gang* iria fazer a atribuição do valor 500 a todos os valores correspondentes ao *array* y. Além do tempo de execução não melhorar, os resultados iriam também ser errados porque na operação $y(i) = 2.0 * x(i)/y(i)$ os cálculos iriam repetir-se nas 4 *threads*. Para resolver este problema é necessário que a diretiva *parallel* seja acompanhada com a diretiva *loop*. Estas duas diretivas são normalmente utilizadas como uma só diretiva sendo estas estruturadas da seguinte forma: *parallel loop*. Com o uso destas diretivas num ciclo o programador declara que é segura a paralelização desse mesmo ciclo e permite que o compilador selecione a organização das iteração do ciclo para determinado acelerador. Com a incrementação da diretiva *loops* é eliminada assim a redundância.

O código seguinte demonstra o possível uso da combinação destas duas diretivas.

```
!$acc parallel loop
do i=1,N
    y(i) = 500
    x(i) = i
end do

!$acc parallel
    !$acc loop
```

```
9       do i=1, N
            y(i) = 2.0 * x(i) / y(i)
11      end do
```

Listing 5.6: Estrutura diretiva parallel loop

Comparação Kernels vs Parallel

Apesar de estas duas diretivas serem parecidas existem algumas diferenças entre elas. A diretiva *kernel* concede ao compilador a máxima liberdade para paralelizar e otimizar o código da forma que este achar mais apropriado para o acelerador em questão, dependendo assim maioritariamente da habilidade do compilador para automaticamente paralelizar o código. A diretiva *parallel loop* por outro lado é uma afirmação feita pelo programador em que declara que é seguro e desejável a paralelização do ciclo em questão. Dependendo assim do programador para julgar e identificar corretamente se alguma parte do código é segura para ser paralelizada, e remover partes do código que sejam inseguras de serem paralelizadas. Se for declarado que o ciclo é paralelizável e este não o for, os resultados obtidos serão incorretos.

Podemos considerar que a diretiva *kernel* é uma dica para o compilador para este analisar a possibilidade de paralelismo, e a diretiva *parallel* consiste numa afirmação do programador da existência da possibilidade de paralelismo. Ao contrário da diretiva *kernels* que também tem a possibilidade de ser acompanhada com a diretiva *loop*, é necessário o uso explícito desta com o uso da diretiva *parallel* quando houver a existência de ciclos.

Para programadores com menos experiência em programação em paralelo, ou para códigos que constituam um grande número de ciclos que necessitam de ser analisados, a opção mais simples será possivelmente a utilização da diretiva *kernels*, sendo que o trabalho de paralelização recairá maioritariamente no compilador. Para programadores com maior experiência em paralelização, que já tenham identificado os ciclos paralelizáveis do seu código o uso da diretiva *parallel* talvez seja a mais indicada. No decorrer da análise da paralelização do código com OpenACC irá ser utilizadas diretivas *parallel loop*.

6

OPENACC - CPU

O foco deste capítulo é o estudo da potencialidade do uso de CPU *multicore* para a paralelização e otimização do código, com recurso ao modelo OpenACC.

6.1 Características do CPU

As execuções do código das secções 3.4.3 e 4.3 foram efetuadas com o recurso a um CPU. O CPU utilizado foi o Intel® Core™ i5-8250U, com as seguintes características:

Tabela 5: Características do CPU utilizado

Arquitetura	x86_64
Byte Order	Little Endian
threadsper Core	2
ncleos per socket	4
socket	1
CPU MHz	848.846
L1d cache	32k
L1i cache	32k
L2 cache	256k
L3 cache	6144k

Todo o código compilado e executado até este capítulo utiliza apenas uma única *thread* do processador no qual está a ser executado. Como discutido na secção 5.1 o número de núcleos e consequentemente *threads* nos processadores tem vindo a aumentar e tem tendência a continuar com o mesmo crescimento durante vários anos. Representado na tabela 5 estão alguns dados característicos do CPU utilizado para a execução e compilação de todo o código realizado ao longo desta dissertação. Este processador, como vários outros presentes no mercado nos tempos de hoje é constituído por quatro núcleos todas elas na mesma *socket*, em que todos eles tem a capacidade

de processar 2 *threads*. Permitindo assim explorar o uso de paralelismo ao nível de *threads* (*TLP*). A exploração de *TLP* pode ser feitas através de dois modelos. Sendo o primeiro o uso de *threads* para pequenas tarefas onde todas têm a mesma finalidade, normalmente chamado de programação em paralelo, e a segunda a execução de várias tarefas todas independentes entre si, que podem até originar mais do que um utilizador, sendo esta chamada de *request-level parallelism*. O modelo estudado será o de programação em paralelo.

6.1.1 *Espaço de memória - CPU*

O desenvolvimento de processadores *multicore* veio permitir que vários núcleos consigam trabalhar em conjunto em tarefas diferentes ou na mesma tarefa. Para que seja possível que estes tenham interação entre si é necessário que estes consigam comunicar e tenham um espaço de memória partilhado. Estes multi-processadores têm o nome de*shared-memory multiprocessors* e dividem-se em duas categorias, dependendo do número de processadores envolventes.

distributed shared memory

A primeira consiste na distribuição física da memória, e tem o nome de *distributed shared memory* (*DSM*). Esta arquitetura é utilizada em grandes números de núcleos em multi-processadores, em que a memória é distribuída entre os processadores, para evitar a grande latência de acesso a uma memória centralizada por todos os processadores. Processadores multicore com menor número de núcleos, como o Intel® Core™ i5-8250U não utilizam este tipo de arquitetura.

Shared-memory multiprocessor

A segunda categoria, sendo esta a presente no CPU com as caracteristicas apresentadas na tabela 5, é intitulada de multi-processadores simétricos de memória partilhada (*SMP*). Esta categoria está normalmente presente em processadores com um menor número de núcleos, tipicamente 8 ou menos núcleos. Os *SMP* devido ao seu número pequeno de núcleos, podem partilhar uma memória centralizada em que todos têm o mesmo nível de acesso à mesma. Com esta arquitetura a necessidade de sincronização de dados entre as memórias de todos os núcleos pode se tornam um impedimento à performance do código, porque por vezes a execução da instrução demora menos tempo do que a sincronização de dados entre as memórias de todos os núcleos. A figura 16 mostra uma possível representação de uma estrutura do tipo *SMP*.

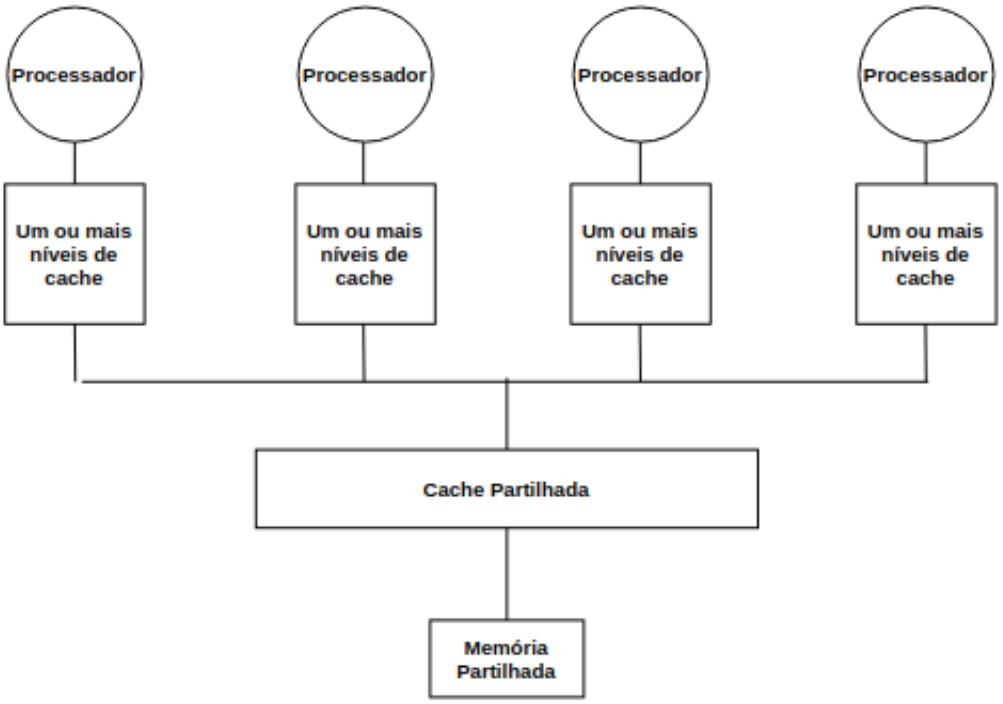


Figura 16: Estrutura SMP

6.2 OPENACC

Nesta secção o código estudado será paralelizado com o recurso às diretivas *parallel loop* já discutidas na secção 5.3.3. A paralelização do código será feita no processador descrito na tabela 5. As paralelizações serão aplicadas nas subdivisões 2,7,8,9,11,14,15,18,19,20,22 e 23 e serão analisados os seus resultados para diferentes números de *threads* utilizados.

Porquê estas subdivisões?

O objetivo inicial é estudar os efeitos da paralelização com o recurso a *threads* em diferentes partes do código. Estas subdivisões foram selecionadas porque é possível paralelizá-las sem a necessidade de alteração do código. Por outro lado, a subdivisão 16 não foi incluída porque apenas é executada dependendo de uma condição, e devido ao seu baixo tempo de execução que não possibilita um estudo tão preciso aos efeitos da paralelização. As restantes subdivisões para serem paralelizadas necessitam de alterações, sendo estas alterações realizadas em capítulos seguintes.

6.2.1 *Diretivas e compilação*

As diretivas utilizadas para realizar a paralelização com recurso ao modelo OpenACC, foram a diretiva *loop* em conjunto com a diretiva *parallel*. Foram utilizadas estas diretivas ao invés da diretiva *kernels* porque os blocos de código a serem otimizados já foram selecionados e estudados de maneira a garantir a possibilidade da sua paralelização.

Estas diretivas foram aplicadas no código com uma estrutura idêntica ao código exemplo seguinte, sendo o valor de N igual ao tamanho de linhas e colunas das matrizes.

```
!$acc parallel loop
    do n = 1 , N
        !$acc loop
        do i=1, N
            C(i) = A(i) + B(i)
        end do
    end do
```

Listing 6.1: Estrutura diretiva loop

O código foi compilado com a *flag* "*-fast*"analisada e discutida na secção 4.3, outras *flags* incrementadas foram:

- -Minfo=accel : Vai instruir o compilador para que imprima o *feedback* sobre o código compilado nas partes que o código foi acelerado pelo OpenACC
- -ta=multicore : Permite ao compilador compilar o código para o "*Target Accelarator* " especificado, neste caso para *threads* num CPU multicore

Após a compilação o compilador imprimiu a seguinte mensagem:

```
$ pgfortran -fast -ta=multicore -Minfo=accel code.f90

    170, Generating Multicore code
        171, !$acc loop gang
    173, Loop is parallelizable
    215, Generating Multicore code
        216, !$acc loop gang
    218, Loop is parallelizable
    230, Generating Multicore code
        231, !$acc loop gang
    233, Loop is parallelizable
    243, Generating Multicore code
        244, !$acc loop gang
    246, Loop is parallelizable
```

```
    266, Generating Multicore code
16      267, !$acc loop gang
    269, Loop is parallelizable
18  291, Generating Multicore code
        292, !$acc loop gang
20  294, Loop is parallelizable
    306, Generating Multicore code
22      307, !$acc loop gang
    309, Loop is parallelizable
24  330, Generating Multicore code
        331, !$acc loop gang
26  333, Loop is parallelizable
    345, Generating Multicore code
28      346, !$acc loop gang
    348, Loop is parallelizable
30  362, Generating Multicore code
        363, !$acc loop gang
32  365, Loop is parallelizable
    381, Generating Multicore code
34      382, !$acc loop gang
    384, Loop is parallelizable
36  394, Generating Multicore code
        395, !$acc loop gang
38  397, Loop is parallelizable
```

Listing 6.2: Compilação Multicore

Esta impressão informa que foi gerado código multicore nos 12 ciclos onde estão presentes as diretivas, sendo que a linha 170 é o início da subdivisão 2, a linha 215 o início da subdivisão 7 e assim sucessivamente. Logo o código foi paralelizado com sucesso em todas estas linhas para ser executado num CPU multicore.

6.2.2 *Tempos de Execução*

OpenACC permite determinar o número de *threads* a serem utilizadas pelo CPU, com o uso da variável de ambiente $ACC\_NUM\_CORES = 2$, neste caso o compilador vai organizar o código para este ser processado em 2 *threads*.

Número de threads: 2

Os tempos de execução das subdivisões otimizadas através de paralelização utilizando o método de compilação discutido na subsecção anterior estão representados na tabela 6, com o uso de 2 *threads* do processador.

Tabela 6: Tempo de cada subdivisão (paralelizada) em 100 iterações - Multicore (2 Threads)

Subdivisão	Tempo em segundos em 100 iterações	Subdivisão	Tempo em segundos em 100 iterações
2	1,47	15	2,9
7	93,8	18	17,86
8	16,2	19	15,63
9	3,6	20	3,71
11	14,97	22	19,3
14	19,3	23	22,14

Sequencial vs 2 Threads

Graficamente colocando lado a lado os valores destas subdivisões paralelizadas para serem executadas em 2 *threads* com os valores dos tempos de execução obtidos através da compilação com o compilador PGI Fortran com diretivas em apenas 1 *thread* obtemos o gráfico da figura 17, com recurso a este gráfico é facilmente observável que todas as subdivisões paralelizadas diminuíram o seu tempo de execução com o recurso a duas *threads* do CPU comparativamente com apenas uma.

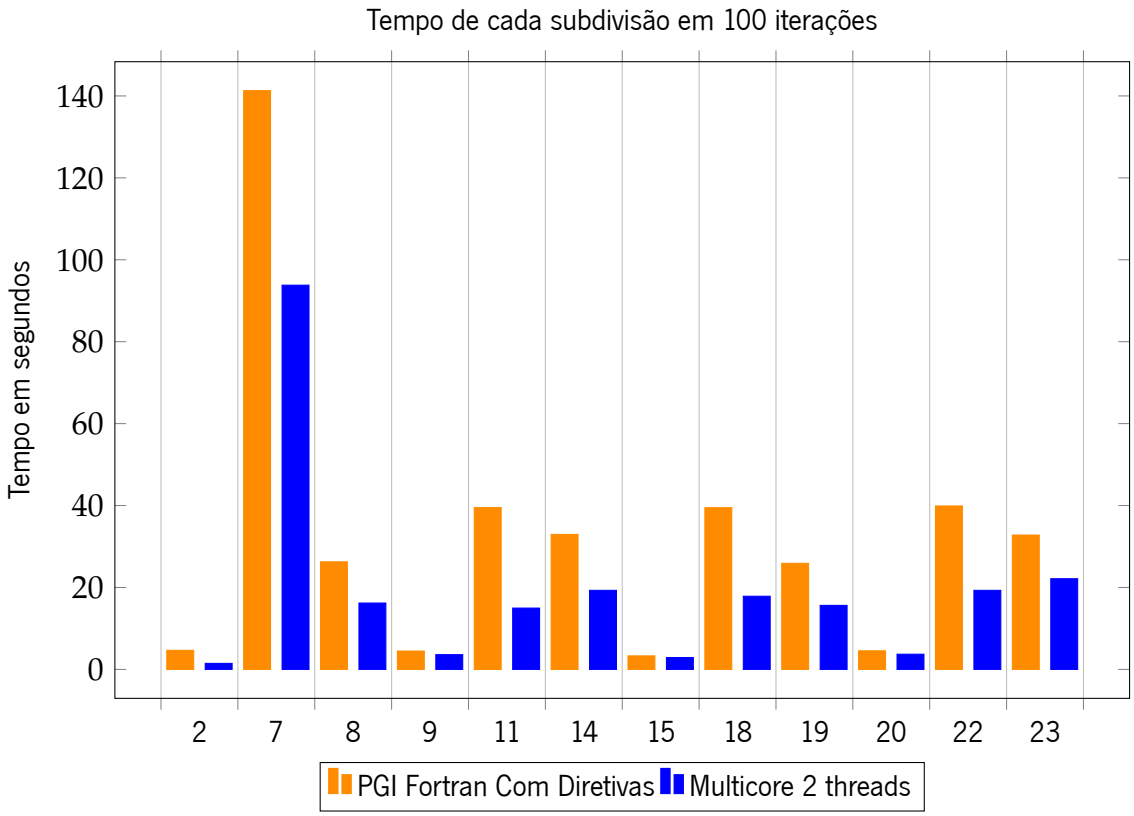


Figura 17: Gráfico de comparação - PGI Multicore(2 Threads) vs PGI Fortran

Transpondo o gráfico para números obtemos o seguinte:

- Subdivisão 2 : Subdivisão 69% vezes mais eficiente com o uso de 2 threads;
- Subdivisão 7 : Subdivisão 34% vezes mais eficiente com o uso de 2 threads;
- Subdivisão 8 : Subdivisão 39% vezes mais eficiente com o uso de 2 threads;
- Subdivisão 9 : Subdivisão 20% vezes mais eficiente com o uso de 2 threads;
- Subdivisão 11 : Subdivisão 62% vezes mais eficiente com o uso de 2 threads;
- Subdivisão 14 : Subdivisão 42% vezes mais eficiente com o uso de 2 threads;
- Subdivisão 15 : Subdivisão 21% vezes mais eficiente com o uso de 2 threads;
- Subdivisão 18 : Subdivisão 55% vezes mais eficiente com o uso de 2 threads;
- Subdivisão 19 : Subdivisão 40% vezes mais eficiente com o uso de 2 threads;
- Subdivisão 20 : Subdivisão 18% vezes mais eficiente com o uso de 2 threads;
- Subdivisão 22 : Subdivisão 51% vezes mais eficiente com o uso de 2 threads;

- Subdivisão 23 : Subdivisão 32% vezes mais eficiente com o uso de 2 threads;
- Todas as subdivisões paralelizadas em soma demoraram 230,88 segundos tendo as apenas compiladas para uso sequencial demorado 395,06 segundos, sendo assim um aumento de eficácia de 41% vezes .

Subdivisões como 2, 11, 18 e 22 obtiveram uma otimização de execução maior do que 2 vezes, isto seria impossível tendo em conta que com o uso do dobro das *threads* o valor máximo do aumento da sua performance em condições ideais seria de apenas 2 vezes. Mas com a transformação dos cálculos das matrizes através de ciclos explícitos o compilador com a *flag -fast* realizou outros tipos de otimização. No caso da subdivisão 2 o compilador não realizou a otimização *MCI*, ao invés gerou um vector *SIMD* para o ciclo. Nas subdivisões 11, 18 e 22 não foi realizada a técnica de *unrolled*, podendo esta ao invés de otimizar o ciclo estar a fazer o contrário, ou seja tornando-o mais lento.

Foram realizadas várias compilações para diferentes números de *threads* de valores entre 2 a 7, para analisar o desempenho do CPU na execução do código para diferentes números de *threads* .

Para uma melhor análise da evolução dos tempos de execução das diferentes subdivisões para diferentes números de *threads* foram realizados diversos gráficos.

O gráfico da figura 18 serve para demonstrar a evolução dos tempos de execução das subdivisões 2, 9, 15 e 20 com o aumento dos números de *threads* utilizados no paralelismo. O gráfico da figura 19 demonstra a evolução da subdivisão 7, o da figura 20 demonstra a evolução das subdivisões 8,11 e 14 e por fim o gráfico da figura 21 demonstra a evolução das subdivisões 18, 19, 22 e 23. As doze subdivisões foram divididas em quatro gráficos para uma melhor compreensão dos mesmos.

Em quase todas as subdivisões existe um decréscimo de tempo de execução entre, uma a quatro *threads*, um aumento na transição de quatro para cinco *threads*e em seguida outro decréscimo de cinco até sete *threads*. A única subdivisão que apresenta um aumento de performance proporcional ao aumento de *threads* é a subdivisão sete. A diminuição de performance na transição entre quatro e cinco *threads* pode dever-se ao facto do tempo necessário para a sincronização dos dados entre todos os processadores aumentar em maior grau do que a diminuição do tempo necessário à execução da subdivisão. Já a subdivisão sete não apresenta essa diminuição provavelmente devido ao seu elevado tempo de execução, que faz com que a otimização feita ao cálculo seja maior do que o aumento do tempo de comunicação entre memórias.

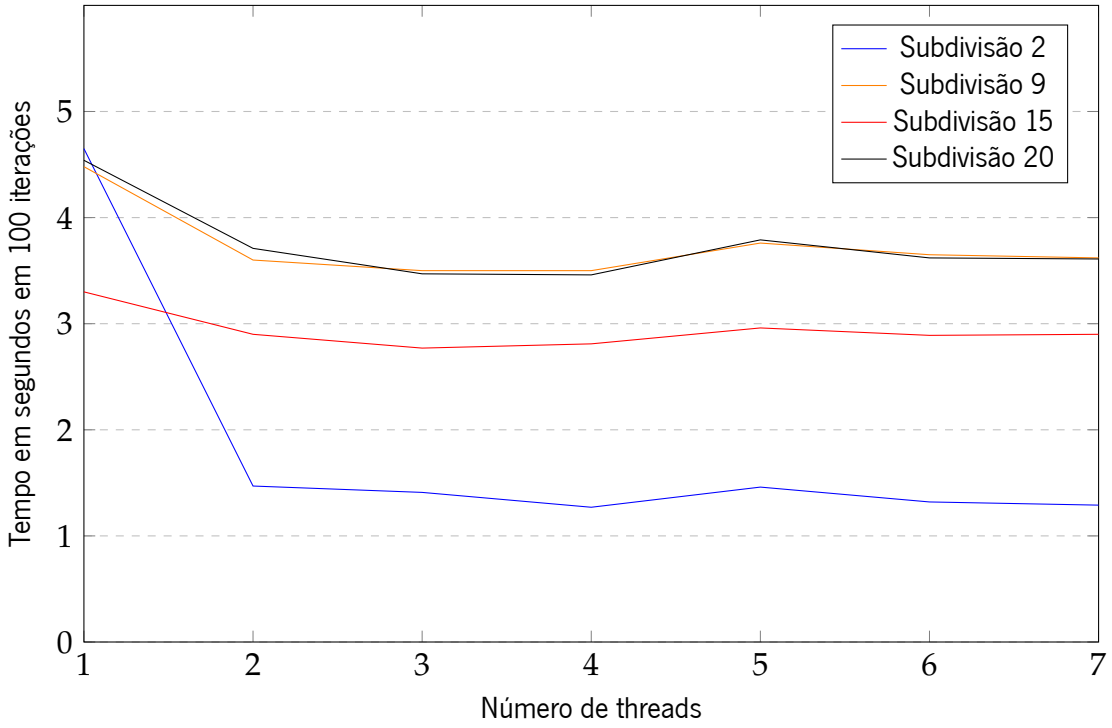


Figura 18: Gráfico da evolução do tempo de execução em relação ao aumento de Threads num CPU - Subdivisões 2,9,15 e 20

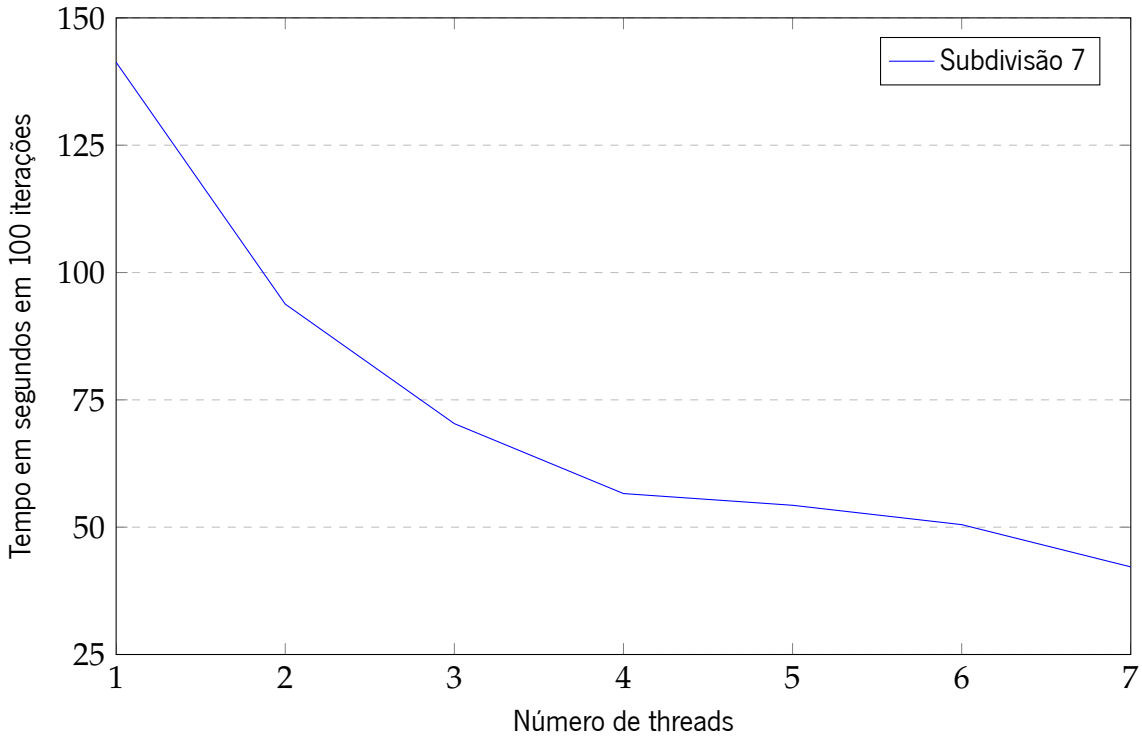


Figura 19: Gráfico da evolução do tempo de execução em relação ao aumento de Threads num CPU - Subdivisão 7

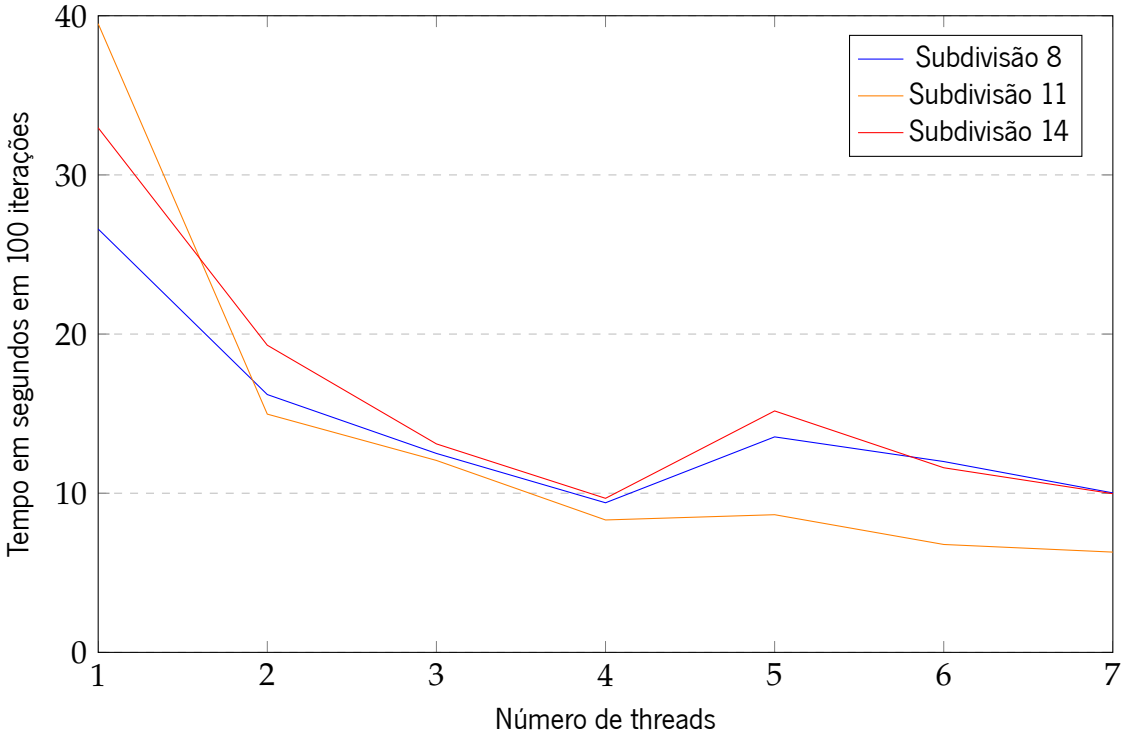


Figura 20: Gráfico da evolução do tempo de execução em relação ao aumento de Threads num CPU - Subdivisões 8,11 e 14

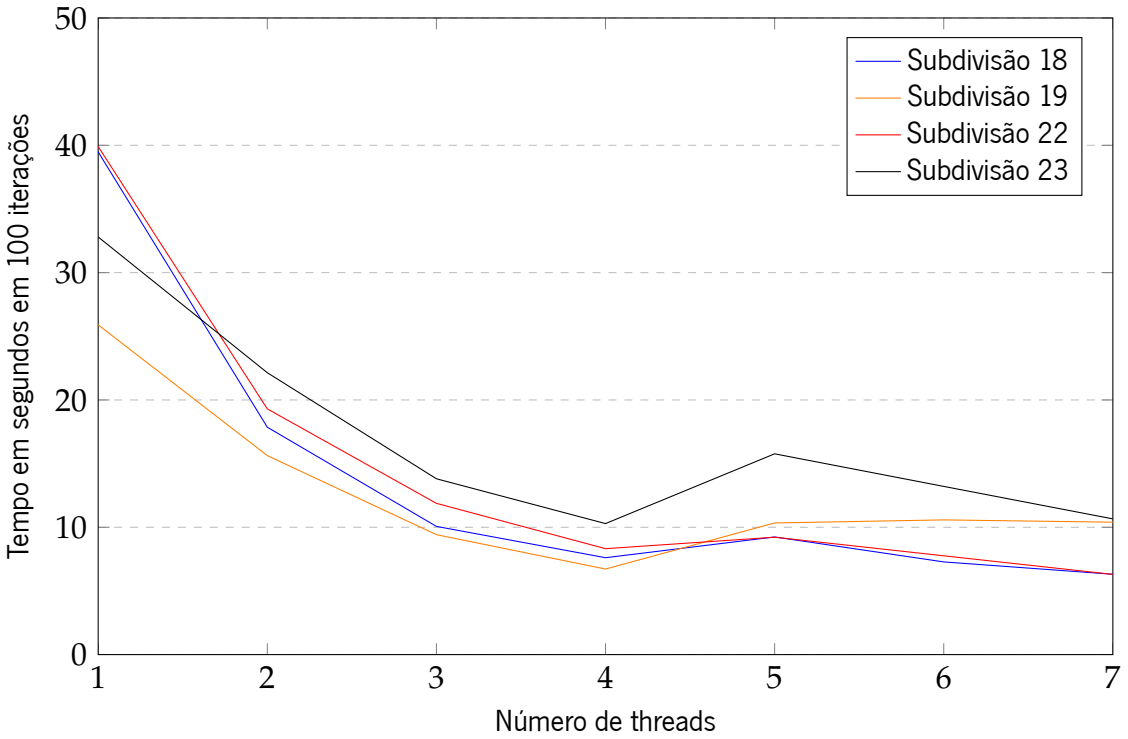


Figura 21: Gráfico da evolução do tempo de execução em relação ao aumento de Threads num CPU - Subdivisões 18,19,22 e 23

7

OPENACC - GPU

Unidades de processamento gráfico (GPUs) tem na generalidade entre 10 a 100 vezes maior poder bruto do que um CPU normal. Apesar de os GPU serem bastante diferentes do e CPUs, sendo que estes consistem num elevado número de unidades de processamento em paralelo, das quais isoladamente são bastante menos potentes do que um único core de um CPU. Logo, para utilizar o verdadeiro poder das unidades de processamento gráfico é necessária uma carga de trabalho que seja altamente paralela. Além disso, os processadores de um GPU processam todos o mesmo código de forma síncrona, logo isto faz com que estes obtenham um elevado desempenho em operações que executem a mesma operação em tipos de dados idênticos.

Em anos mais recentes, GPUs têm-se tornado cada vez mais uma opção bastante viável para realizar tarefas de processamento em paralelo, em vez de serem apenas limitadas a processamento gráfico. Apesar de, o método utilizado para a programação de GPUs ser diferente dos métodos utilizados para programar CPUs devido ao diferente conjunto de instruções de cada arquitectura. Por isso, é necessário um compilador especialmente desenhado para ir de encontro aos requisitos necessários para compilar um código desenvolvido para um GPU.

Com as diretivas de OpenACC, como visto anteriormente, é possível paralelizar o código. O compilador PGI em conjunto com o modelo OpenACC permite que este código paralelizado consiga ser executado tanto em CPU como em GPU.

Primeiramente irá ser feita uma análise mais intuitiva e superficial sobre o funcionamento do GPU com o uso do modelo OpenACC.

7.1 Memórias - CPU vs GPU

A maior partes dos computadores comercializados na atualidade são constituídos por um CPU e um GPU. O CPU é onde corre o sistema operativo e é nele que os programas começam a ser executados. O CPU tem um pequeno número de núcleos mas um grande espaço de memória devido a esta ser relativamente barata. Já o GPU possui centenas de núcleos, apesar de serem núcleos mais simples do que as do CPU o seu número é bastante maior. A velocidade de transferência de dados entre o

espaço de memória e os registos é também mais rapida no GPU do que no CPU, implicando assim que este espaço de memória seja mais caro tornando com que este tenha uma menor capacidade.

A comunicação entre o CPU e o GPU é realizada através de um conetor de entrada e saída, portanto qualquer transferência de dados realizada entre os dois será através deste conetor. A velocidade relativa entre a velocidade de transmissão do conetor e da memória do GPU é bastante grande, sendo a velocidade do conetor bastante mais lenta. Mesmo que a velocidade do GPU seja muito rápida a comunicar com a memória das *thread blocks* se os dados em seguida forem enviados para o CPU através do conetor será gasto um elevado tempo de computação na comunicação entre os dois processadores, relativamente ao tempo de execução das instruções. Devido à velocidade de transmissão do conetor ser um entrave devem-se evitar ao máximo comunicações desnecessárias entre o CPU e o GPU. Na figura 22 está representado o que se deve esperar de uma máquina que tenha um CPU e um GPU, a largura das setas representa a velocidade de comunicação entre as memórias. (As setas não estão à escala)

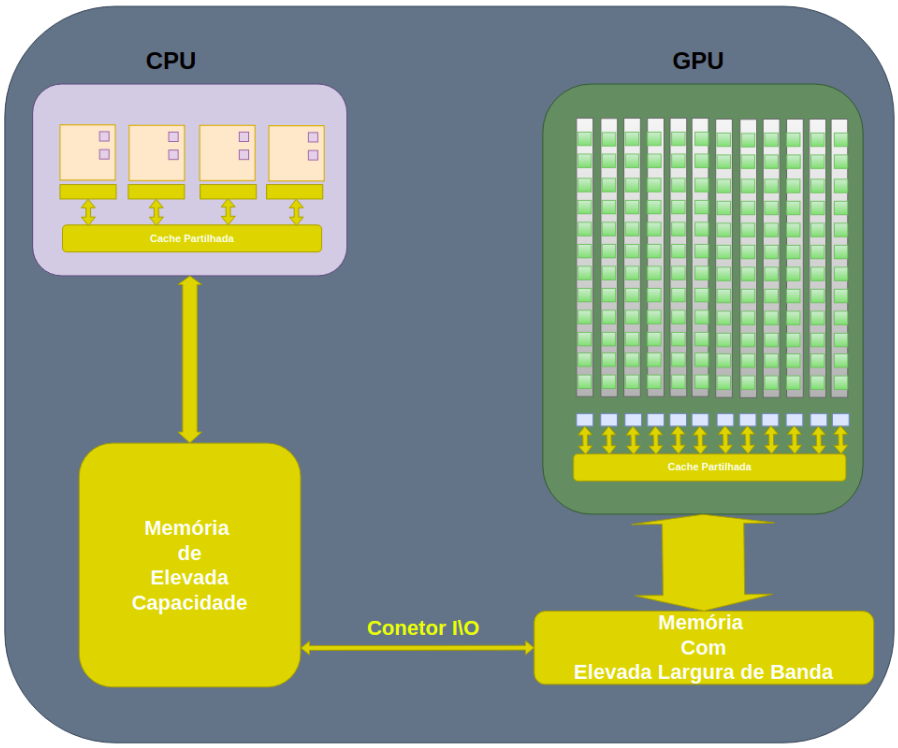


Figura 22: Diagrama CPU + GPU

7.2 CUDA unified memory

Unified Memory é uma tecnologia que basicamente pega na parte esquerda da figura 23 que representa a organização física dos dois processadores e faz com que esta se pareça com a parte direita da mesma figura. Ou seja, faz com que pareça que ambos os processadores tenham em comum um

grande espaço de memória físico. Sendo este processo normalmente chamado *managed memory*. Este processo tem o nome de *managed memory* porque sem qualquer influência do programador os dados estão a ser geridos da memória de um processador para o outro pelo sistema operativo e pelo driver do dispositivo.

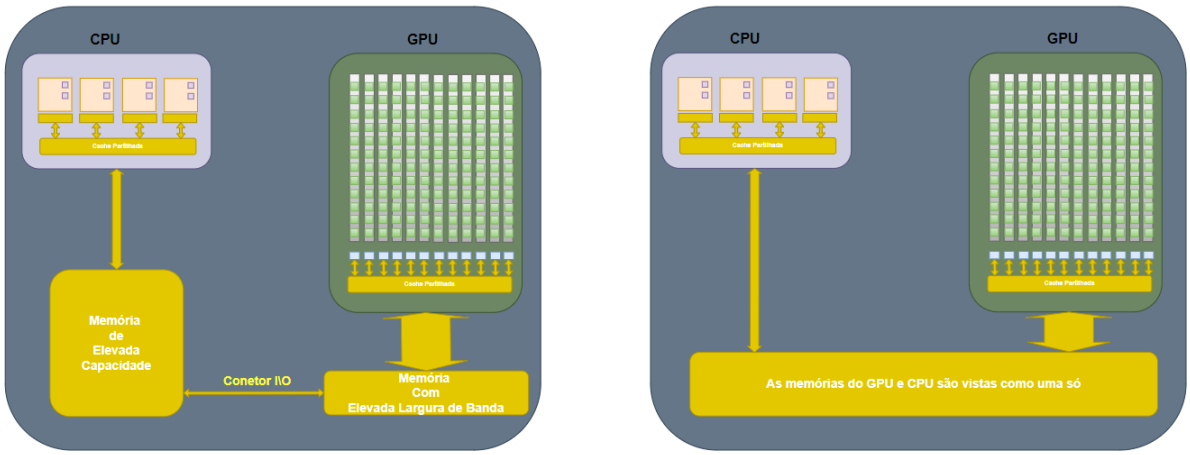


Figura 23: Managed Memory

Esta é uma ferramenta que possibilita aceder à memória do GPU sem a necessidade de obter conhecimentos mais profundos acerca da mesma, ou seja não é necessária a preocupação com a movimentação dos dados porque quem lida com isso é a ferramenta. O uso desta ferramenta possibilita ao programador que se concentre mais no paralelismo e pensar apenas na movimentação de dados como uma otimização. O compilador PGI consegue utilizar *managed memory* com a ativação de *flags*. Este recurso será estudado na secção seguinte. Isto são tudo vantagens para utilizar a ferramenta, mas esta também apresenta algumas limitações, como por exemplo:

- Com o uso de *managed memory* a transferência de dados é obrigatoriamente síncrona, isto pode trazer algumas limitações ao código para os casos em que é necessário, ou benéfico a transferência de dados assincronamente
- Alocação e libertação de memória demora mais tempo com o uso de *managed memory*, isto pode tornar-se num problema se existirem muitas alocações durante a execução do código
- Esta ferramenta ainda só está disponível para o compilador PGI em GPU da Nvidia
- Na maior parte das vezes realizar as transferências de dados manualmente melhora a performance do código. Quanto maior o controlo do programador sobre o que está a decorrer com o programa, maior a probabilidade de este o conseguir otimizar

7.3 Managed memory - OpenACC

Como discutido na secção anterior o uso da ferramenta *managed memory* pode ser bastante útil na programação de GPUs. Nesta secção será feito o estudo do desempenho desta ferramenta na compilação do código referente a esta dissertação. O código a ser compilado é o mesmo que o compilado na secção 6.2. O compilador PGI permite ativar o uso desta ferramenta através da flag $-ta = tesla : managed$ sendo que o "$tesla$"serve para identificar a placa gráfica utilizada, neste caso uma Nvidia e o *managed* para indicar a utilização da ferramenta *managed memory*.

7.3.1 *Compilação*

Para a compilação foi utilizada a *flag* $-Minfo = accel$ para obter os dados de aceleração realizados pelo compilador e o resultado foi o seguinte:

```
    170, Generating Tesla code
        171, !$acc loop gang ! blockidx%x
        173, !$acc loop vector(128) ! threadidx%x
        170, Generating implicit copyout(rr_coord(1:jj,1:jj)) [if not
    already present]
         Generating implicit copyin(y_coord(1:jj,1:jj),x_coord(1:jj,1:
    jj)) [if not already present]
    173, Loop is parallelizable
    215, Generating Tesla code
        216, !$acc loop gang ! blockidx%x
        218, !$acc loop vector(128) ! threadidx%x
    215, Generating implicit copy(pfftin(:,:)) [if not already present]
         Generating implicit copyin(y_coord(:,:),x_coord(:,:)) [if not
    already present]
    218, Loop is parallelizable
                                .
                                .
                                .
    362, Generating Tesla code
        363, !$acc loop gang ! blockidx%x
        365, !$acc loop vector(128) ! threadidx%x
    362, Generating implicit copyin(apc(1:jj,1:jj)) [if not already
    present]
         Generating implicit copy(pfftin(1:jj,1:jj)) [if not already
    present]
         Generating implicit copyin(exp_pi_i_r2_lam_ddel(1:jj,1:jj)) [
    if not already present]
    365, Loop is parallelizable
    381, Generating Tesla code
```

```
         382, !$acc loop gang ! blockidx%x
25       384, !$acc loop vector(128) ! threadidx%x
     381, Generating implicit copyin(pfftout(:,:)) [if not already
    present]
27        Generating implicit copyout(pfftin(:,:)) [if not already
    present]
     384, Loop is parallelizable
29   394, Generating Tesla code
         395, !$acc loop gang ! blockidx%x
31       397, !$acc loop vector(128) ! threadidx%x
     394, Generating implicit copyin(ds_arr(j),pfftin(1:jj,1:jj)) [if
    not already present]
33        Generating implicit copy(id_r(1:jj,1:jj)) [if not already
    present]
     397, Loop is parallelizable
```

Listing 7.1: Compilação Managed Memory

As partes do código paralelizadas foram as mesmas da secção 6.2, isto podendo ser confirmado pelo número da linha em que a paralelização foi feita nesta compilação comparativamente com a anterior. Sabemos que a paralelização foi feita para o código ser executado num GPU pelas declarações "Generating Tesla code"presente em todas as subdivisões paralelizadas. As restantes mensagens serão discutidas e analisadas em secções posteriores. A informação importante a retirar é que todos os ciclos foram paralelizadas com sucesso para serem executadas no GPU, apenas com a modificação de uma *flag*.

7.3.2 *Tempos de execução*

Com o uso do compilador PGI e a ferramenta *managed memory* os valores de execução de cada subdivisão e movimentação de dados ligados às mesmas foram os representados na tabela 7.

Obtiveram-se os seguintes valores com esta execução:

- Tempo total dos 100 ciclos : 737,92 segundos , cerca de 15% mais lento do que a execução sequencial com o compilador PGI e aproximadamente 40% vezes mais lento do que a execução paralelizada com 3 *threads* no CPU
- As subdivisões paralelizadas em conjunto demoraram 376,97 segundos. Aproximadamente 67% mais lento do que a execução com 4 *threads* no CPU;
- Tempo total da Divisão 2 : 205,1 segundos
- Tempo total da Divisão 3 : 523,81 segundos

- Comparativamente com a execução sequencial utilizando PGI, apenas a subdivisão 7 diminuiu o seu tempo de execução em aproximadamente 69%

Tabela 7: Tempo de cada subdivisão em 100 iterações - GPU Managed Memory

Subdivisão	Tempo em segundos em 100 iterações	Subdivisão	Tempo em segundos em 100 iterações
1	0,0007	13	19,49
2	7,77	14	18,39
3	0,77	15	38,55
4	0,77	16	0,000015
5	185,88	17	43,1
6	4,85	18	36,86
7	43,85	19	18,1
8	39,89	20	38,89
9	39,89	21	42,79
10	38,17	22	36,87
11	39,93	23	17,98
12	20,06		

Os valores obtidos na execução do código para GPU com o uso da ferramenta *managed memory*, foram bastantes insatisfatórios. À primeira análise, das 12 subdivisões apenas a subdivisão 7 obteve melhorias, e todas as outras pioraram o seu desempenho.

7.3.3 *Análise com PGI Profiler*

PGI Profiler (PGPROF) é um programa da empresa PGI[37] que permite uma melhor análise ao desempenho de uma aplicação. PGPROF fornece uma forma para visualizar e diagnosticar o desempenho de todos os componentes do programa. Este programa utiliza gráficos e tabelas para associar os tempos de execução com o código e instruções do programa, permitindo assim a visualização de onde os tempos de execução são gastos. É possível utilizar PGPROF para analisar programas que utilizem paralelização, tanto em CPU com em GPU.

Para uma melhor compreensão da execução do código compilado na subsecção anterior, este será analisado com o recurso ao programa PGPROF.

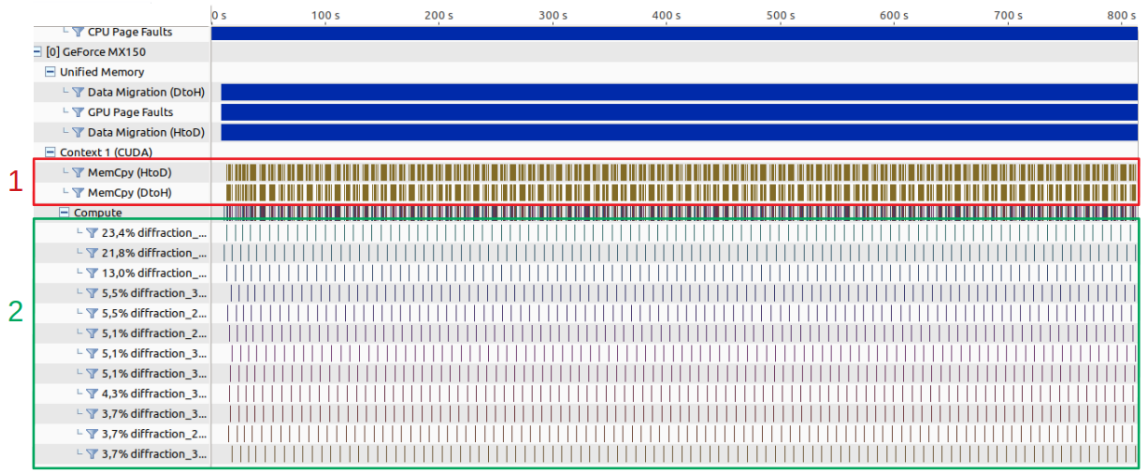


Figura 24: PGI profiler Managed memory

Na figura 24 está presente a análise total da execução do programa com o uso de *managed memory*. Na zona 1 representada a vermelho na figura estão representados os tempos de *MemCpy (HtoD)* que são os tempos de transferência dos dados da memória do CPU para a memória do GPU e em baixo os tempos de *MemCpy (DtoH)* que são os tempos de transferência de dados do GPU para o CPU. A zona 2 representada a verde corresponde aos tempos de execução das 12 subdivisões paralelizadas e executadas pelo GPU. A uma primeira análise é observável pela figura que existe uma maior presença da cor dourada, que representa as transferências de memória, do que das núcleos referentes à execução dos cálculos de cada subdivisão. São visíveis também as barras a azul, em que a número 3 representa os *CPU Page Faults* ou seja as vezes em que o CPU necessitava de aceder à memória que não estava presente no seu espaço de memória e o mesmo caso para o GPU representado no número 4. Este problema é bastante recorrente com a utilização da ferramenta *managed memory* em que a memória apenas é transferida após o primeiro pedido dos processadores.

Para ser mais perceptível a análise realizada pelo programa PGI Profiler ao código, foi reduzido o *timestamp* para uma iteração representado na figura 25. Nesta figura estão representadas a cor vermelha os espaços em que as execuções das subdivisões foram realizadas pelo CPU, sendo essa a razão pela qual não aparecem as barras representativas do tempo de execução.

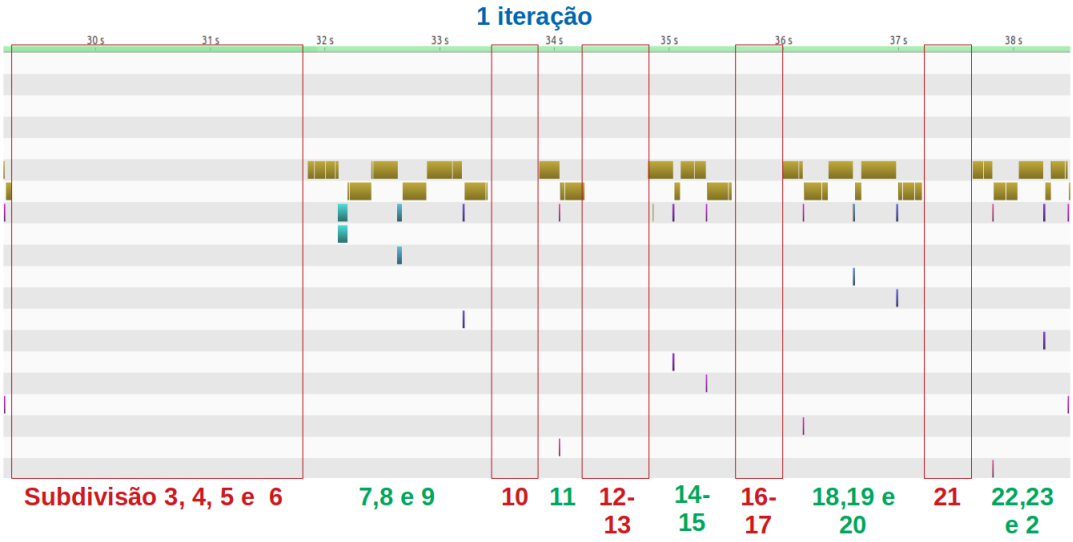


Figura 25: PGI profiler Managed memory

Com a ajuda da figura 25 é possível perceber que todas as vezes que é realizada a execução de um cálculo numa subdivisão no GPU existe a movimentação de dados entre os dois processadores, o que consome bastante tempo de execução. Em todos os casos o tempo da transferência de dados é maior do que o próprio tempo de execução dos cálculos. Sendo assim o tempo de comunicação e transferência de dados entre os dois processadores um *bottleneck*. Na tabela 7 os tempos de cada subdivisão são respetivos aos tempos de execução dos cálculos mais o tempo de transferência de dados. Para estudar a influência do tempo perdido na transferência dos dados entre as memórias dos dois processadores foi criada a tabela 8 onde se encontra tudo discriminado.

Tabela 8: Tempo de cada subdivisão em 100 iterações - GPU Managed Memory

Subdivisão	Tempo de cálculo em 100 iterações	Tempo de transferência de dados em 100 iterações	Tempo total em segundos em 100 iterações
1	0,0007	0	0,0007
2	1,3	6,4	7,7
3	0,77	0	0,77
4	0,77	0	0,77
5	185,88	0	185,88
6	4,85	0	4,85
7	8,2	35,65	43,85
8	4,5	35,39	39,89
9	1,7	37,19	38,89
10	38,17	0	38,17
11	1,3	38,63	39,93
12	20,06	0	20,06
13	19,49	0	19,49
14	1,7	16,69	18,39
15	1,4	37,15	38,55
16	0,000015	0	0,000015
17	43,1	0	43,1
18	1,3	35,56	36,86
19	1,9	16,2	18,1
20	1,8	37,09	38,89
21	42,79	0	42,79
22	1,4	35,47	36,87
23	1,7	16,28	17,98
Total :	385,03	348,78	732,86

7.3.4 *Conclusões - Managed Memory*

Com recurso ao programa PGI Profiler foi possível analisar de forma mais profunda o comportamento do código ao ser compilado com o recurso à ferramenta *managed memory*. Apesar desta ferramenta permitir ao programador compilar e executar o seu código em dois processadores diferentes sem que este se tenha que preocupar com a transferência de dados entre os dois, esta pode não ser uma opção viável quando o código não se encontra 100% paralelizado, como é o caso. Os cálculos

realizados tornaram-se mais rápidos ao serem executados no GPU mas devido à necessidade da movimentação constante dos dados entre os dois processadores o tempo de execução total foi 15% mais demorado do que com a execução do código totalmente sequencial. O tempo consumido pela transferência de dados foi aproximadamente 47.2%, sendo então de grande importância diminuir este valor para aumentar o desempenho do código. O tempo total que o GPU demorou a executar as 12 subdivisões paralelizadas foi de aproximadamente 28.2 segundos ou seja cerca de 282 milissegundos por iteração, sendo estes valores incrivelmente baixos comparando com os valores obtidos com o uso de um CPU.

7.4 Gestão de memória

Como discutido na secção anterior, é de elevada importância ser o programador a controlar as transferências e a utilização de memória dos processadores. Sem a necessidade de utilizar a ferramenta *managed memory* não só se elimina a dependência de utilizar o compilador PGI e Nvidia GPUs como se elimina outros erros como o de a memória por vezes chegar demasiado tarde apenas após o pedido do processador, originando *Page Faults*.

7.4.1 *Análise de Variáveis por subdivisão*

É necessário estudar que variáveis são necessárias para a execução de cada cálculo, em cada subdivisão e se estas são alteradas ou não na execução do cálculo. Foi desenvolvido o diagrama da figura 26 similar ao da figura 2 onde as variáveis correspondentes a matrizes de 4096 colunas por 4096 linhas estão representadas por números inseridos em círculos, nas subdivisões em que são utilizadas. No total, na secção 3 são utilizadas 17 variáveis com as características referidas. Quando a variável é apenas de saída, ou seja, o seu valor anterior não é necessário para a realização anterior mas irá ser necessário nas subdivisões seguintes, a sua cor atribuída será a vermelha. Se a variável é de entrada, ou seja, se o seu valor é utilizado na subdivisão mas não é alterado na mesma, a sua cor respetiva será a verde. Por último, se a variável for necessária para a execução da subdivisão e for alterada no decorrer da mesma a sua cor atribuída será a azul.

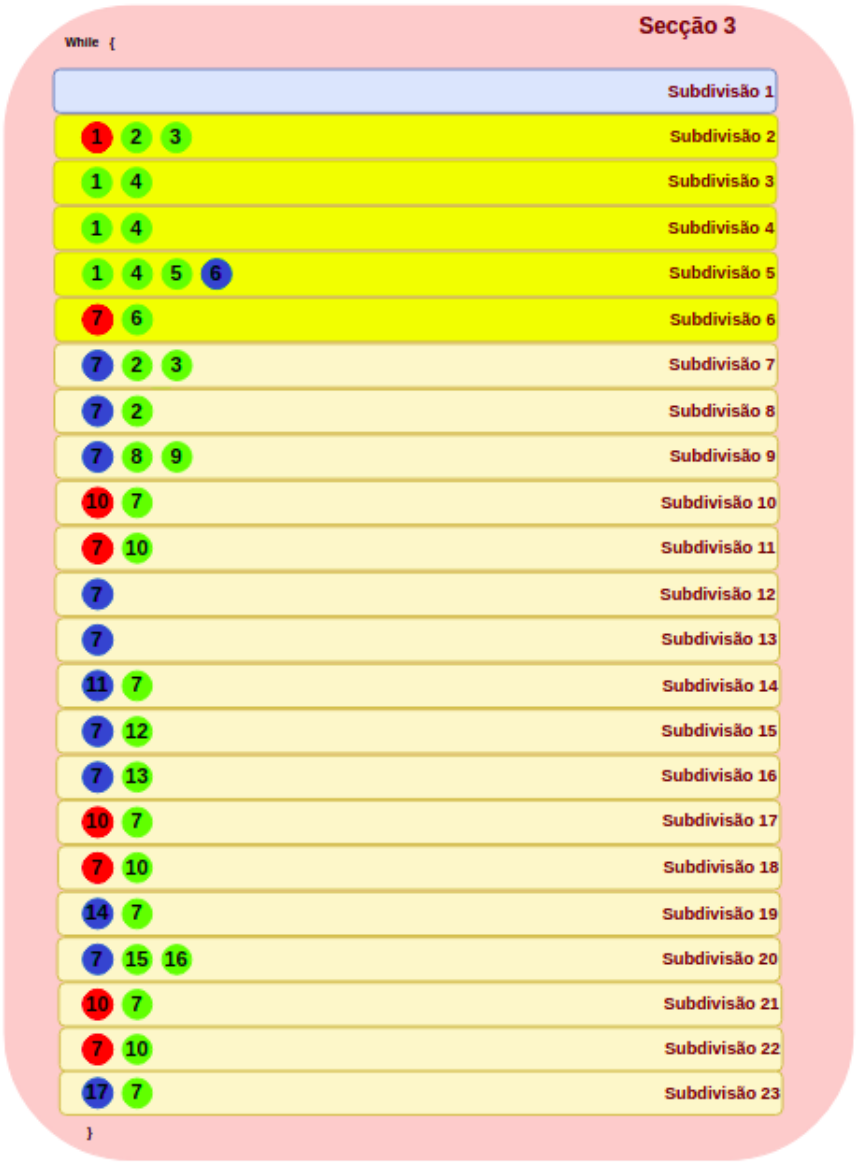


Figura 26: Variáveis correspondentes a cada subdivisão

7.4.2 *Diretiva - Data Clauses*

OpenACC tem diretivas que permitem a gestão de dados. Estas dizem ao compilador para criar código que execute movimentação de dados especifica e fornece dicas sobre o uso desses dados. Estas diretivas permitem ao programador que não esteja dependente do uso da ferramenta *managed memory* e consiga ser ele a decidir quando a movimentação de dados irá ser feita e quais os dados a serem utilizados.

A estrutura da diretiva em Fortran é a seguinte:

```
!$acc data clauses
        .
        .
        .
!$acc end data
```

Listing 7.2: Estrutura diretiva data

Podendo as *clauses* ser as seguintes:

- copy
- copyin
- copyout
- create

Copy

Esta cláusula aloca memória no GPU e copia memória do CPU para o GPU quando entra dentro da região criada pela cláusula e quando sai copia os dados de volta para o CPU. Esta cláusula é utilizada para dados que necessitam ser de entrada e de saída. Sendo o seguinte exemplo prático:

```
!$acc data copy(C(1:N,1:N))
    !$acc parallel loop
    do n = 1 , N
        !$acc loop
        do i=1, N
            C(n,i) = 500 + C(n,i)
        end do
    end do
!$acc end data
```

Listing 7.3: Estrutura diretiva copy

Neste exemplo o *array C* necessita de ser copiado do CPU para o GPU e em seguida retornado porque irá ser feito um incremento de 500 a todos os valores do *array*.

Copyout

Esta cláusula aloca memória no GPU e copia memória do GPU para o CPU na região final da cláusula. Esta cláusula é utilizada para dados que necessitam ser de saída mas não necessitam de ser de entrada. Sendo o seguinte exemplo prático:

```
!$acc data copyout(C(1:N,1:N))
    !$acc parallel loop
    do n = 1 , N
        !$acc loop
        do i=1, N
            C(n,i) = 500
        end do
    end do
!$acc end data
```

Listing 7.4: Estrutura diretiva copyout

Neste exemplo não é necessário copiar os valores do *array C* apenas alocá-los, porque serão todos substituídos pelo valor 500, sendo só necessário no final da região limitada pela cláusula copiar o valor de volta para o CPU.

Copyin

Esta cláusula aloca memória no GPU e copia memória do CPU para o GPU na região inicial da cláusula. Esta cláusula é utilizada para dados que necessitam ser de entrada mas não necessitam ser de saída. Sendo o seguinte exemplo prático:

```
!$acc data copyout(C(1:N,1:N)) copyin(A(1:N,1:N))
    !$acc parallel loop
    do n = 1 , N
        !$acc loop
        do i=1, N
            C(n,i) = A(n,i)
        end do
    end do
!$acc end data
```

Listing 7.5: Estrutura diretiva copyin

Neste exemplo não é necessário copiar os valores do *array C* apenas alocá-los, porque serão todos substituídos pelo valor do *array a*. É necessário alocar e copiar inicialmente os valores do *array a* mas não é necessário retorná-los pois estes não foram modificados. Só é necessário no final da região limitada pela cláusula copiar o valor do *array C* de volta para o CPU.

Com o uso destas cláusulas é possível controlar quando e quais as movimentações de dados serão realizadas, eliminando assim a dependência da ferramenta *managed memory*.

Organização da transferência de variáveis

É importante recordar que apenas as subdivisões 2,7,8,9,11,14,15,18,19,20,22,23 estão a ser executadas no GPU. Com o auxílio do diagrama da figura 26 é possível de melhor forma organizar o modo como a transferência de dados será realizada. Para uma melhor compreensão sobre como foi realizada a organização da transferência de dados entre o CPU e o GPU foram adicionadas às 23 subdivisões existentes novas subdivisões, com o nome de subdivisões de transferência de dados. Estas novas subdivisões representam as cláusulas discutidas na secção 7.4.2. Estas cláusulas serão inicializadas antes de uma subdivisão que será executada no GPU e encerrada antes de uma subdivisão que será executada no CPU, excepto entre a subdivisão 1 e a 2 onde não existe essa necessidade, devido à subdivisão 1 não manipular nenhuma matriz. Sendo o código organizado da forma representada na figura 27.

Sendo que os dados transferidos em cada subdivisão de transferência de dados (STD) são os seguintes:

- **STD 1** - *copyout* : Variável 1, *copyin* : Variável 2 e 3
- **STD 2** - *copy* : Variável 7, *copyin* : Variável 2, 3,8 e 9;
- **STD 3** - *copyout* : Variável 7, *copyin* : Variável 10
- **STD 4** - *copy* : Variável 7 e 11, *copyin* : Variável 12
- **STD 5** - *copy* : Variável 14, *copyin* : Variável 10,15 e 16 , *copyout* : 7
- **STD 6** - *copy* : Variável 17, *copyin* : Variável 10, *copyout* : 7

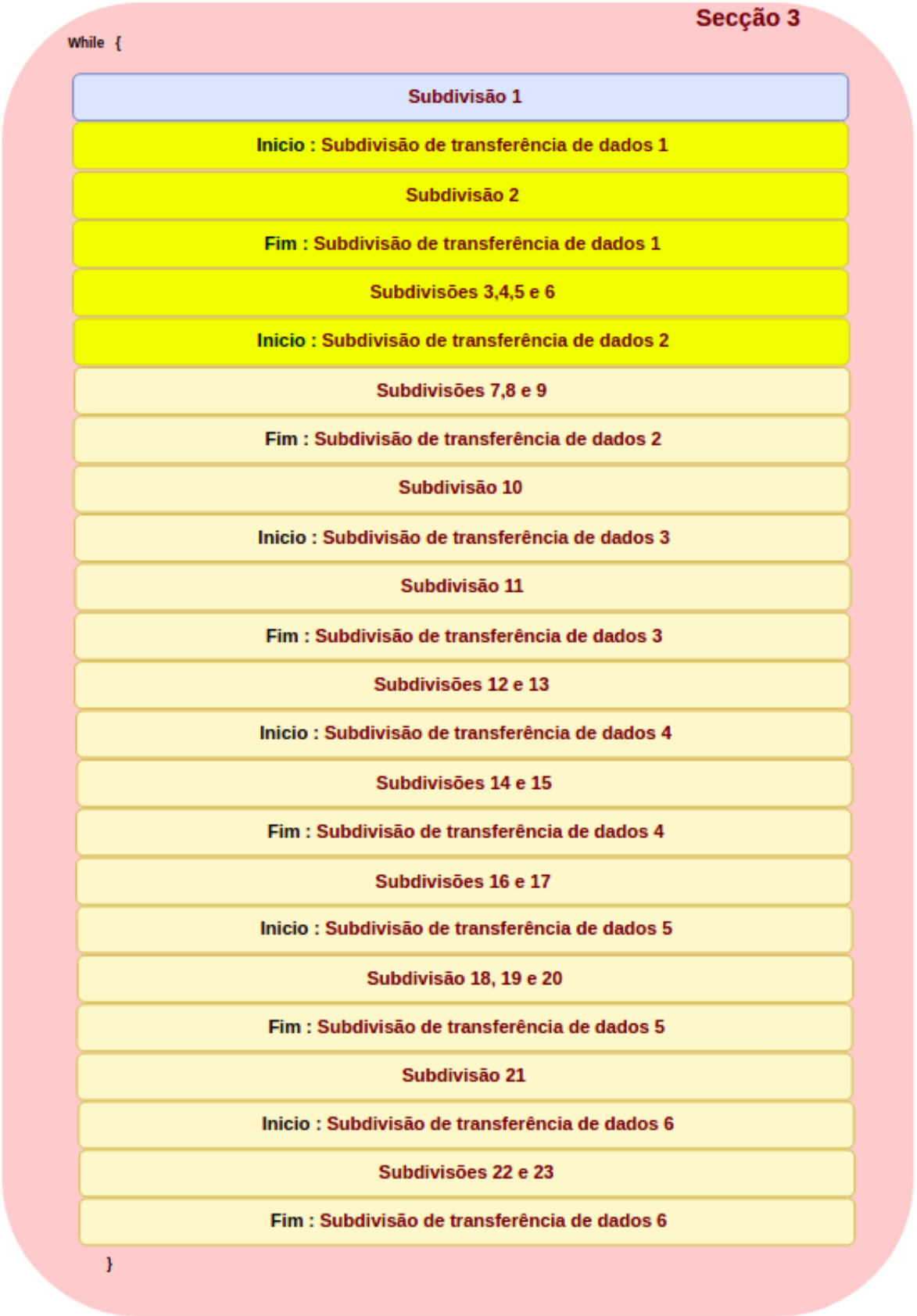


Figura 27: Variáveis correspondentes a cada subdivisão

Compilação

Para esta compilação foram utilizadas as *flags*, $-Minfo = accel$ e $-ta = tesla$, com as diretivas *data copyin* copyin, *copyout* e *copy* nas subdivisões do código discutidas acima. Sendo o resultado imprimido pelo compilador o seguinte :

```
1     172, Generating copyin(y_coord(1:jj,1:jj),x_coord(1:jj,1:jj)) [if
    not already present]
          Generating copy(rr_coord(1:jj,1:jj)) [if not already present]
    // copyout
3     174, Generating Tesla code
          175, !$acc loop gang ! blockidx%x
5         177, !$acc loop vector(128) ! threadidx%x
      177, Loop is parallelizable
7     215, Generating copyin(exp_pi_i_r2_lamz0(1:jj,1:jj)) [if not
    already present]
          Generating copy(pfftin(1:jj,1:jj)) [if not already present]
9         Generating copyin(y_coord(1:jj,1:jj),x_coord(1:jj,1:jj),apa(1:
    jj,1:jj)) [if not already present]
      220, Generating Tesla code
11        221, !$acc loop gang ! blockidx%x
          223, !$acc loop vector(128) ! threadidx%x
13    223, Loop is parallelizable
                              .
15                            .
                              .
17    400, Generating copy(id_r(1:jj,1:jj)) [if not already present]
          Generating copyout(pfftin(1:jj,1:jj)) [if not already present]
19        Generating copyin(pfftout(1:jj,1:jj)) [if not already present]
      406, Generating Tesla code
21        407, !$acc loop gang ! blockidx%x
          409, !$acc loop vector(128) ! threadidx%x
23    409, Loop is parallelizable
      418, Generating Tesla code
25        419, !$acc loop gang ! blockidx%x
          421, !$acc loop vector(128) ! threadidx%x
27    421, Loop is parallelizable
```

Listing 7.6: Compilação diretivas data

Os dados imprimidos pelo compilador confirmam que as transferências de memória estão a ser efetuadas nos locais certos. Comparativamente com a impressão de compilação da secção 7.3.1, por exemplo, na linha 215 do código são feitas mais operações de *copyin*, sendo estas as descritas acima. De notar também que nesta compilação nenhuma transferência de dados foi realizada de

forma implícita pelo compilador, porque todas as variáveis foram declaradas corretamente no código ao invés de ter de ser o compilador a fazer esse trabalho implicitamente. O número das linhas de código paralelizadas modificaram um pouco desde a ultima compilação devido ao incremento das *data clauses*.

Tempos de execução

Com o uso do compilador PGI e das diretivas de gestão de dados do OpenACC, os valores obtidos para todas as subdivisões representadas na figura 27 estão demonstrados na tabela 9.

Sintetizando, com este ensaio foram obtidos os seguintes dados:

- Tempo total dos 100 ciclos: 633,6 segundos, ainda aproximadamente 2% mais lento do que com a execução sequencial com o compilador PGI e aproximadamente 30% mais lenta do que com a execução paralelizada com 3 *threads* do CPU;
- As subdivisões paralelizadas em conjunto demoraram 289,09 segundos com o acréscimo dos tempos de transferência de dados. Aproximadamente 55% lento do que a execução com 4 *threads* no CPU;
- Tempo total da Divisão 2 : 211,68 segundos;
- Tempo total da Divisão 3 : 421,92 segundos;
- 91 % do tempo de execução das subdivisões paralelizadas do código é utilizado na transferência de dados, sendo apenas 9% utilizado para fazer os cálculos no GPU

Tabela 9: Tempo de cada subdivisão em 100 iterações - GPU OpenACC Data clauses

Subdivisão	Tempo de cálculo em 100 iterações	Tempo de transferência de dados em 100 iterações	Tempo total em segundos em 100 iterações
1	0,00002	0	0,00002
STD 1	0	20,15	20,15
2	1,34	0	1,34
3	0,9	0	0,9
4	0,9	0	0,9
5	184,16	0	184,16
STD 2	0	54,77	54,77
6	4,23	0	4,23
7	8,06	0	8,06
8	4,53	0	4,53
9	1,77	0	1,77
10	39,18	0	39,18
STD 3	0	34,53	34,53
11	1,3	0	1,3
12	18,07	0	18,07
13	18,24	0	18,24
STD 4	0	48,57	48,57
14	1,67	0	1,67
15	1,46	0	1,46
16	0,000047	0	0,000047
17	36,74	0	36,74
STD 5	0	57,85	57,85
18	1,3	0	1,3
19	1,67	0	1,67
20	1,77	0	1,77
21	42,09	0	42,09
STD 6	0	45,36	45,36
22	1,3	0	1,3
23	1,69	0	1,69
Total :	372,37	261,23	633,6

Análise com PGI Profiler

Para uma análise mais meticulosa foi utilizada a aplicação PGI Profiler. O resultado obtido está representado na figura 28.

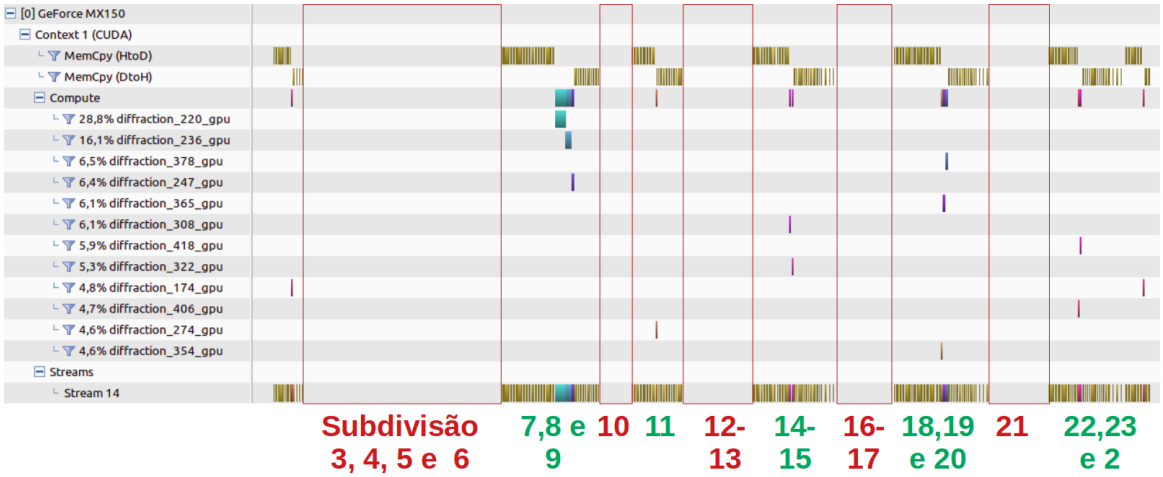


Figura 28: PGI Profiler- Data Clauses

Com recurso a esta análise é possível confirmar que a movimentação de dados está a ser efetuada como pretendida. No espaço designado pelas subdivisões paralelizadas, representadas a verde, existem apenas duas movimentações como esperado. Sendo a primeira com o nome de *MemCpy HtoD*, referindo-se à transferência de dados do CPU para o GPU e às suas respetivas alocações. E a segunda *MemCpy DtoH*, referindo-se à transferência de dados do GPU para o CPU. Este método de agrupar as subdivisões paralelizadas que são sequências permitiu diminuir bastante o tempo e o número de vezes que as transferências de dados são efetudas entre os dois processadores. Com a utilização de PGI Profiler é também claro que o erro de *Page Faults*, que representa as vezes em que um dos processadores tentou aceder a um dado que não estava presente na sua memória, foi eliminado e não voltou a acontecer. Porque com o uso de *data clauses* é possível transferir os dados necessários antes que o processador tente fazer o acesso aos mesmos.

Conclusões do código compilado

O tempo total da execução dos 100 ciclos continua a ser bastante mais baixo com a utilização de *threads* do CPU. Isto deve-se ao facto do elevado tempo necessário para a comunicação entre os dois processadores. Com o auxílio da diretiva *data clauses* foi possível diminuir o número de vezes em que esta comunicação é realizada, diminuindo assim consequentemente o tempo de execução do código. O tempo de execução das operações nas subdivisões continua a ser incrivelmente mais rápido com o uso do GPU. Com a gestão manual de quando e quais as variáveis a ser transferidas e alocadas o tempo total de execução diminuiu em aproximadamente 18% em relação aos tempos obti-

dos com a ferramenta *managed memory*. Sendo que os tempos relacionados com as subdivisões paralelizadas diminuiu em aproximadamente 23%.

Com os resultados obtidos nesta secção é possível verificar que apesar da ferramenta *managed memory* permitir ao programador a abstração àcerca da transferência de dados, esta não retira o proveito do código a 100%. Sendo que ocorreram melhorias significativas com as transferências de dados serem organizadas forma não implícita.

7.4.3 *Diretiva Sincronização de dados - Update*

Diminuir ou eliminar por completo o tempo causado pela transferência de dados é essencial para a otimização do código. Como recurso da diretiva *data* foi possível eliminar algum do tempo perdido desnecessariamente com a movimentação dos dados entre os dois processadores. Nesta subsecção serão estudados os efeitos de aplicação de uma outra diretiva de gestão de dados de OpenACC, sendo esta uma diretiva de sincronização de dados. A diretiva *data* permite transferir dados entre processadores mas esta operação é sempre acompanhada com a operação de alocação de dados na memória do processador destinatário. Sendo que, os dados estão apenas alocados para o intervalo definido pela diretiva *data*. Criando um pequeno problema de otimização, porque as *n* vezes que for necessário movimentar os dados de um processador para o outro será também necessário alocar os mesmos *arrays n* vezes. Uma solução para resolver esse problema é a utilização da diretiva *update*. A diretiva *update* de OpenACC informa o compilador que é necessário criar código que realize a sincronização das variáveis entre o CPU e o GPU. Esta diretiva só é válida dentro da região criada pela diretiva *data* sendo que, a sincronização de um *array* só pode ser efetuada se ambos os processadores tiverem esse *array* alocado nos seus espaços de memória.

A estrutura da diretiva *update* em Fortran é a seguinte:

```
!$acc data create(a(1:N,1:N))

    !$acc update clause(a(1:N,1:N))

!$acc end data
```

Listing 7.7: Estrutura diretiva Update

Podendo a *clause* ser:

- Self : Faz com que a sincronização dos dados entre os dois processadores seja feita com os dados do CPU;
- Device : Faz com que a sincronização dos dados entre os dois processadores seja feita com os dados do GPU;

Começar e encerrar uma região referente à diretiva *data* requer bastante poder computacional, sendo mais eficiente atualizar os dados entre o CPU e o GPU com a diretiva *update*.

Reorganização de transferência de memória

Apenas as variáveis representadas com os números 1,7,10,11,14 e 17 são alteradas no decorrer das operações realizadas na secção 3, como se pode observar na figura 26. Sendo que o resto das variáveis não sofrem qualquer tipo de alteração não existe necessidade de em todas as iterações transferir os dados das mesmas entre os processadores, sempre que estas são necessárias. Com o recurso da diretiva *update* é possível encontrar uma solução para este problema. Esta solução está representada no diagrama seguinte:

Figura 29: PGI Profiler- Data Clauses

Foram incluídas duas novas subdivisões exteriores ao ciclo principal da secção 3 do código. Sendo a primeira subdivisão exterior, inserida anteriormente ao início do ciclo, em que a sua função é a de alocar e transferir todas as variáveis necessárias para a memória do GPU. Com a primeira divisão pretende-se evitar o tempo de alocação constante posteriormente gasto dentro do ciclo. Das treze variáveis que são necessárias para as funções correspondentes as subdivisões paralelizadas apenas três destas são necessárias para as subdivisões que correm no CPU, fazendo com que as outras dez não necessitem de ser atualizadas no GPU durante o processamento do ciclo. E a segunda subdivisão exterior, inserida posteriormente ao fim do ciclo, em que a função desta é de fazer a

atualização dos valores de três variáveis na memória do CPU. Estas três variáveis serão as únicas a ser utilizadas nas secções seguintes do código. Dentro do ciclo as subdivisões de início e fim de transferência de dados foram substituídas quase na totalidade por subdivisões de *update* (SU) de variáveis. Como referido anteriormente, com a utilização da diretiva *update* pretende-se atualizar apenas as variáveis necessárias sem ter de obrigatoriamente voltar a fazer a sua alocação nas memórias do processador. As subdivisões *update* estão representadas com duas cores. A cor vermelha representa as situações em que a atualização de dados é feita com os valores do GPU para o CPU e a cor verde representa as situações em que a atualização de dados é feita com os valores do CPU para o GPU.

Compilação

Para esta compilação foram utilizadas as *flags*, $-Minfo = accel$ e $-ta = tesla$, com as diretivas *data* e *update* nas subdivisões do código discutidas anteriormente. Sendo o resultado imprimido pelo compilador o seguinte :

```
    160, Generating copyout(io_r(1:jj,1:jj)) [if not already present]
         Generating create(rr_coord(1:jj,1:jj)) [if not already present
]
         Generating copyin(y_coord(1:jj,1:jj),x_coord(1:jj,1:jj)) [if
not already present]
         Generating create(pfftout(1:jj,1:jj)) [if not already present]
         Generating copyin(exp_pi_i_r2_lam_ddel(1:jj,1:jj)) [if not
already present]
         Generating create(pfftin(1:jj,1:jj)) [if not already present]
         Generating copyout(id_r(1:jj,1:jj)) [if not already present]
         Generating copyin(exp_pi_i_r2_lamz0(1:jj,1:jj),apc(1:jj,1:jj),
apo(1:jj,1:jj),apa(1:jj,1:jj)) [if not already present]
         Generating copyout(ic_r(1:jj,1:jj)) [if not already present]
    176, Generating Tesla code
        177, !$acc loop gang ! blockidx%x
        179, !$acc loop vector(128) ! threadidx%x
    179, Loop is parallelizable
    188, Generating update self(rr_coord(1:jj,1:jj))
    215, Generating update device(pfftin(1:jj,1:jj))
    221, Generating Tesla code
        222, !$acc loop gang ! blockidx%x
        224, !$acc loop vector(128) ! threadidx%x
    224, Loop is parallelizable
                                .
                                .
                                .
    376, Generating Tesla code
```

```
        377, !$acc loop gang ! blockidx%x
25      379, !$acc loop vector(128) ! threadidx%x
    379, Loop is parallelizable
27  387, Generating update self(pfftin(1:jj,1:jj))
    397, Generating update device(pfftout(1:jj,1:jj))
29  401, Generating Tesla code
        402, !$acc loop gang ! blockidx%x
31      404, !$acc loop vector(128) ! threadidx%x
    404, Loop is parallelizable
33  413, Generating Tesla code
        414, !$acc loop gang ! blockidx%x
35      416, !$acc loop vector(128) ! threadidx%x
    416, Loop is parallelizable
```

Listing 7.8: Compilação Diretivas data e update

Com a mensagem de compilação enviada pelo compilador, é visível que o código continua a ser paralelizado para GPU nas subdivisões pretendidas. As primeiras doze linhas desta mensagem correspondem à subdivisão de transferência e alocação de dados, localizada anteriormente ao início do ciclo. Estas 12 linhas informam que foram apenas alocadas as variáveis 1,7 e 10 com a diretiva *data* e a *clause create*, as variáveis 11,14,17 são alocadas e definidas como variáveis de saída em que a sua transferecia de dados de volta para o CPU é realizda na subdivisão *update* 11, e as variáveis restantes são alocadas e definidas como variáveis de entrada pois o seu valor não vai ser alterado durante o ciclo. O compilador imprimiu tambem mensagens que dão a informação que a atualização irá ser realizada na linha de código definida, como por exemplo a mensagem *188, Generating update* $self(rr_coord(1:jj,1:jj))$ que corresponde à subdivisão *update* 1.

Tempos de execução

Após feita a compilação, o código foi executado e os valores obtidos foram os presentes na tabela 10:

Tabela 10: Tempo de cada subdivisão em 100 iterações - Final

Subdivisão	Tempo de cálculo em 100 iterações	Tempo de transferência de dados em 100 iterações	Tempo total em segundos em 100 iterações
STAD	0	0,58	0,58
1	0,00001	0	0,00001
SU 1	0	5,8	5,8
2	1,36	0	1,36
3	0,9	0	0,9
4	0,9	0	0,9
5	184,74	0	184,74
6	4,52	0	4,52
SU 2	0	16,7	16,7
7	8,12	0	8,12
8	4,51	0	4,51
9	1,77	0	1,77
SU 3	0	20	20
10	40,16	0	40,16
SU 4	0	16,19	16,19
11	1,3	0	1,3
SU 5	0	20	20
12	17,56	0	17,56
13	17,65	0	17,65
SU 6	0	16,6	16,6
14	1,69	0	1,69
15	1,46	0	1,46
SU 7	0	19,33	19,33
16	0,000019	0	0,000019
17	37,56	0	37,56
SU 8	0	16,2	16,2
18	1,31	0	1,31
19	1,69	0	1,69
20	1,78	0	1,78
SU 9	0	20,1	20,1
21	44,13	0	44,13
SU 10	0	16,8	16,8
22	1,32	0	1,32
23	1,69	0	1,69
SU 11	0	0,316	0,316
Total :	375,81	168,61	544,42

Resumindo foram obtidos os seguintes dados:

- Tempo total dos 100 ciclos: 544,4 segundos, aproximadamente 12% mais eficiente do que com a execução sequencial com o compilador PGI e aproximadamente 18% menos eficiente do que com a execução paralelizada com 3 *threads* do CPU
- As subdivisões paralelizadas em conjunto demoram 195,72 segundos com o acréscimo dos tempos de transferência de dados. Aproximadamente 35% menos eficiente do que a execução com 3 *threads* no CPU, aproximadamente 55% vezes mais eficiente de que com o uso da ferramente *managed memory*
- Tempo total da Divisão 2 : 214,92 segundos
- Tempo total da Divisão 3 : 328,92 segundos
- 86% do tempo de execução das subdivisões paralelizadas do código é utilizado na transferência de dados, sendo apenas 14% utilizado para fazer os cálculos no GPU

Análise com PGI Profiler

Foi analisada a execução do código com o programa PGI Profiler para uma melhor visualização e entendimento da movimentação dos dados realizadas entre os dois processadores, estando o resultado apresentado na figura 30.

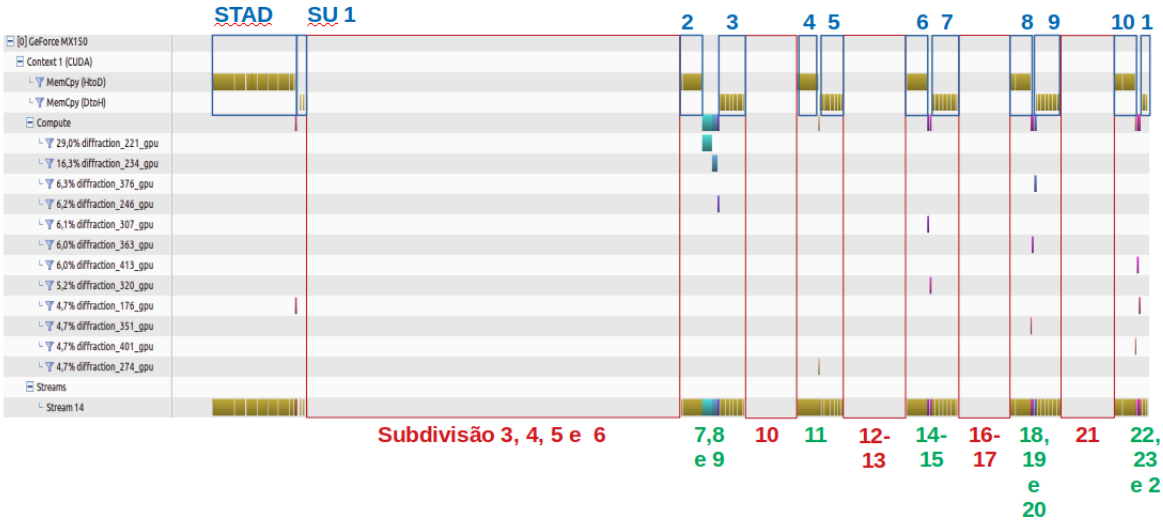


Figura 30: PGI Profiler- Data Clauses

Com o auxílio da análise realizada é possível identificar os momentos em que as operações foram efetuadas e quando os dados foram atualizadas e alocados nas memórias dos processadores. Os números a vermelho representam as subdivisões do código que foram executadas pelo CPU e os

números a verde representam as subdivisões executadas pelo GPU tal como nas figuras analisadas anteriormente. Na figura 30 foram incrementados os números de cor azul que representam as subdivisões que são responsáveis pela transferência de dados entre as memórias de ambos os processadores. Na figura está representada a primeira iteração do ciclo e ainda é observável a subdivisão de transferência e alocação de dados (STAD), que é realizada antes do ciclo. O erro de *Page Faults* continua eliminado, significando assim que os dados estão a chegar a tempo e corretamente às memórias dos dois processadores.

7.5 Comparação de resultados

Foram realizados três ensaios diferentes durante este capítulo. Sendo o primeiro ensaio correspondente à compilação do código com o recurso da ferramenta *managed memory*, disponível pelo compilador PGI. Esta ferramenta pode ser bastante útil principalmente se todo o código estiver já preparado para ser totalmente otimizado. Com o uso da mesma não é necessária a preocupação com a movimentação de memória entre os processadores, sendo que permite ao programador que se foque maioritariamente no algoritmo. Esta ferramenta tem também algumas desvantagens, sendo uma delas o recorrente erro de *Page Faults* que corresponde à memória "chegar"sempre atrasada aos processadores. Uma outra desvantagem é que, são realizadas demasiadas transferências de memória entre os compiladores se o código não estiver totalmente paralelizado. Desde este primeiro ensaio foi possível constatar o elevado desempenho do GPU na realização de cálculo com matrizes, mas também o problema que este mesmo acarreta, o excessivo tempo para a memória ser transferida entre os processadores.

No segundo ensaio, com a diretiva *data* tornou-se possível a paralelização para GPU sem a dependência da ferramenta *managed memory*. A diretiva *data* permitiu que as transferências fossem controladas manualmente, mas sempre com a necessidade dessas transferências serem acompanhadas com uma alocação dos dados a serem transferidos. Com o uso desta, também foi eliminado o erro de *Page Faults*, porque a memória era sempre alocada e transferida antes de uma instrução ser executada. Apesar de, com o uso da diretiva *data* os resultados tenham melhorado devido à diminuição das vezes em que eram alocados e transferidos os dados, as variáveis não alteradas em qualquer tipo de operação continuam a ser alocadas e transferidas na iteração seguinte.

No terceiro e último ensaio, foi incrementada a diretiva *update* à já existente diretiva *data*. Com a utilização desta nova diretiva foi possível eliminar todo o tempo perdido a alocar memória e fazer com que variáveis não alteradas durante os cálculos não fossem novamente transferidas na iteração seguinte. Com a incrementação desta diretiva o tempo de comunicação de dados entre ambos os processadores foi reduzido significativamente. Apesar da eliminação de transferências desnecessárias, 30% do tempo de execução do código é ocupado pelas transferências de dados necessárias. O tempo de execução de todos os cálculos paralelizados em soma no decorrer das 100 iterações,

foram de aproximadamente 28 segundos no GPU e de aproximadamente 128 segundos com o uso de 4 *threads* no CPU. O que demonstra a grande capacidade de velocidade de cálculo de grandes matrizes no GPU.

No gráfico da figura 31 estão ilustrados os tempos ocupados pela comunicação de dados entre os dois processadores nos três ensaios realizados. No eixo horizontal estão representadss os três ensaios sendo o primeiro ensaio com a utilização da ferramenta *managed memory* representado pelas letras MM, o segundo ensaio onde foi utilizadas as diretiva *data* representado pelas letras DD e por fim o terceiro ensaio onde foram utilizadas a diretiva *data* em conjunto com a diretiva *update* representado pela letra DDU. Com este gráfico é possível perceber uma diminuição quase linear do tempo ocupado nas transferências de memória.

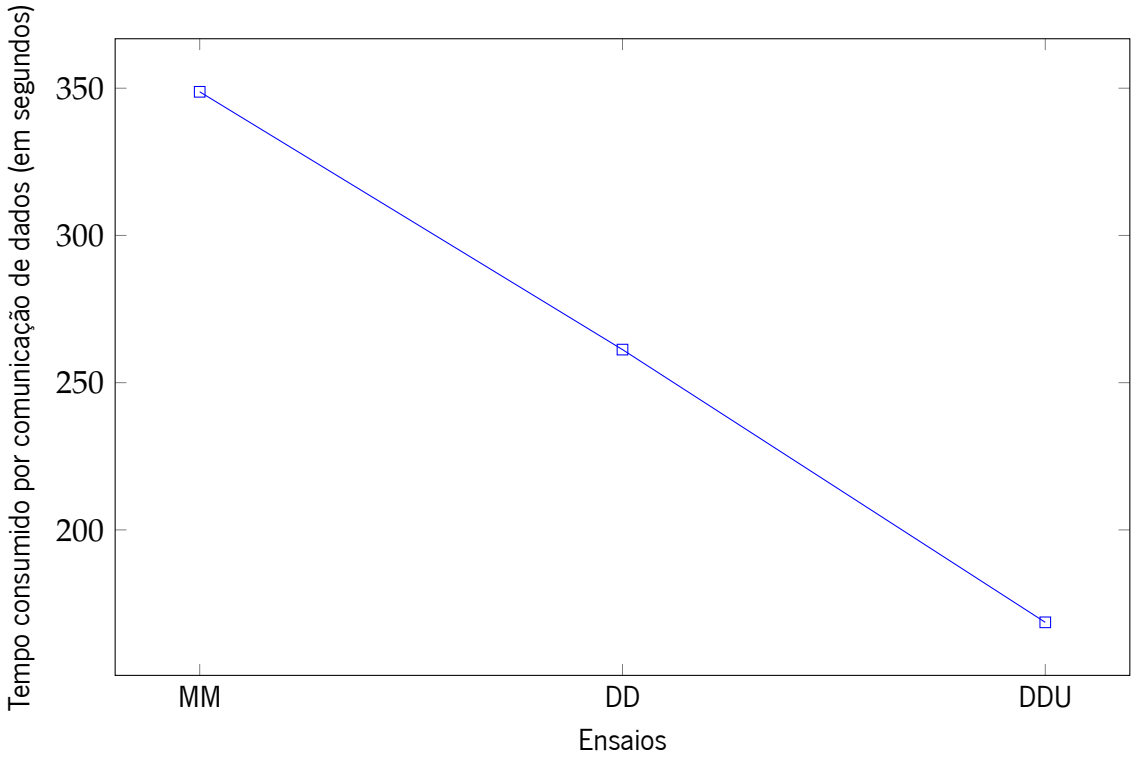


Figura 31: Tempo de transferência de dados com e sem gestão de dados

8

CUDA

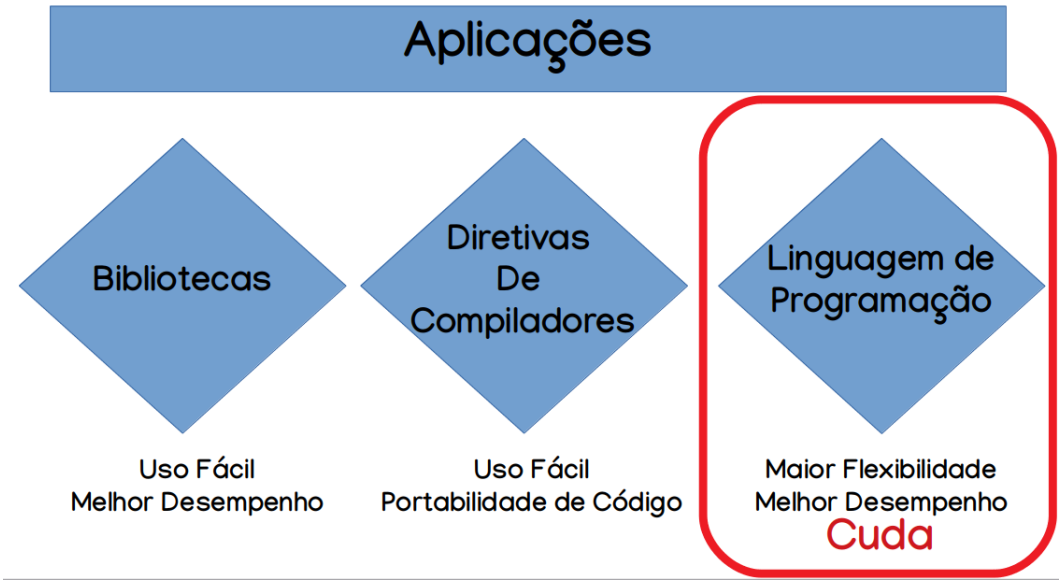


Figura 32: Três Formas de otimizar- CUDA

8.1 Introdução

Os GPU de hoje em dia oferecerem vários recursos tanto para processamento gráfico como não gráfico, tendo como exemplo os resultados já obtidos no capitulo anterior. Devido a este facto, Nvidia desenvolveu CUDA [38], uma nova arquitetura de hardware e software para realizar e gerir execuções paralelas no GPU. Esta linguagem produz código Fortran/ C / C++ para o CPU(*host*) e produz também código com as mesmas linguagens para o GPU(*host*, daí o D em CUDA). Uma linguagem de programação semelhante ao CUDA é OpenCL, estaconsiste em várias empresas desenvolver uma linguagem que possa ser processada em diferentes plataformas.

CUDA é uma *framework* que tem como intuito trabalhar em ambientes maioritariamente paralelos. Esta considera que o GPU tem a sua própria memória, ao contrário por exemplo da ferramenta *managed memory*, logo para serem realizadas tarefas na unidade de processamento gráfica é

necessário que os dados sejam transferidos entre a mesma e o *host*. CUDA fornece acesso geral ao endereçamento de memória da DRAM, logo é possível escrever e ler dados localizados na DRAM tal como com um CPU.

O objetivo de reprogramar o código com CUDA é o de paralelizar a secção três do código por completo para evitar o custo gerado pela transferência de dados, e aumentar a sua performance.

8.2 Arquitetura

Quando programado com CUDA, o GPU é visto como um dispositivo de computação capaz de executar um grande número de *threads* em paralelo. O GPU opera como um coprocessador do CPU principal (*host*).

As *threads* do GPU estão organizadas em blocos dentro de uma grelha, como representado na figura 33.

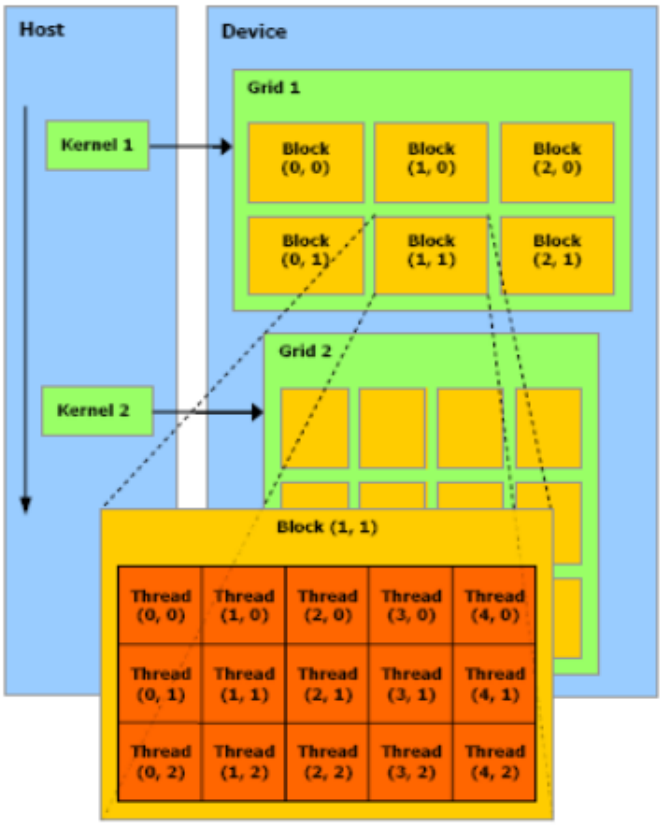


Figura 33: Representação de threads

Um bloco de *threads* é constituído por várias *threads* que têm a capacidade de cooperar umas com as outras, compartilhando dados com eficácia através de memória partilhada de alta velocidade e sincronizando eficazmente as suas execuções para uma melhor coordenação a aceder à

memória. Cada *thread* dentro do bloco de *threads* é identificada pelo seu *thread* ID. Existe um número máximo de *threads* que um bloco pode ter. Os blocos podem ser agrupados numa grelha de blocos. O GPU deve executar todos os blocos de uma grelha sequencialmente se existir baixa capacidade de paralelismo, ou em paralelo se existir uma grande capacidade de paralelismo, ou até uma combinação dos dois. Cada bloco dentro da grelha é identificado pelo seu *block* ID. A constituição do modelo de memória de uma grelha está representada na figura 34. Os *registers* e a *local memory* apenas podem ser lidos ou alterados por cada *thread*, a *shared memory* é a memória partilhada entre *threads*, a *global memory* é a memória partilhada entre blocos e pode ser alterada e as *constant* e *texture memory* são partilhadas entre os blocos da mesma grelha mas apenas podem ser lidas.

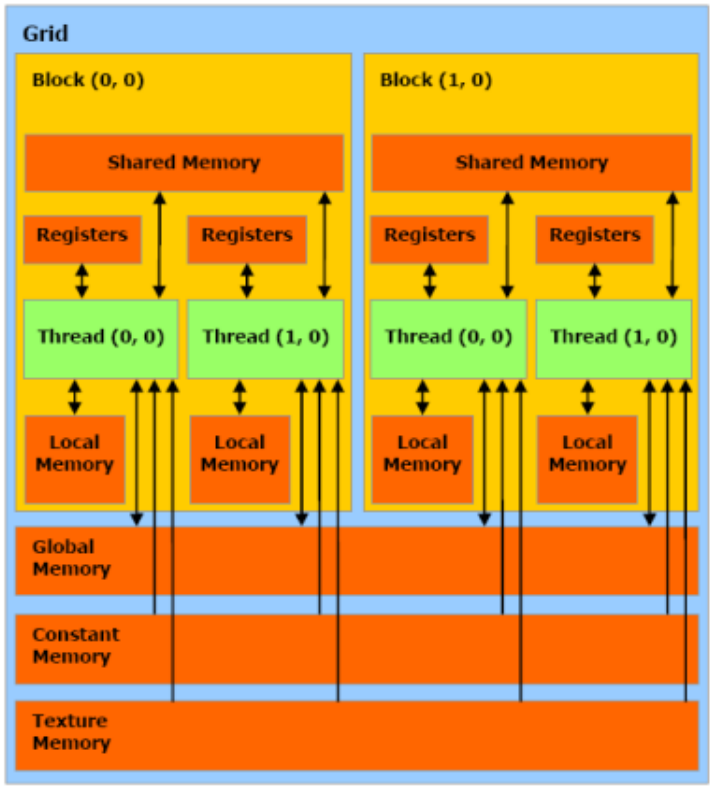


Figura 34: Representação do modelo de uma grelha de *threads*

8.3 Princípios de programação com cuda

Para programar com CUDA é necessário primeiramente entender alguns detalhes como:

- Para distinguir funções que serão executadas no GPU (device) de funções executadas no CPU(host), CUDA utilizada "__device__"ou "__global__"para o GPU e "__host__"para o CPU

- Variáveis declaradas em funções com "__device__"ou "__global__"são alocadas na memória global do GPU
- A sintaxe para a chamada de uma função que é executada no GPU é, nome<<dimGrelha, dimBloco>>(parâmetros), onde dimGrelha e dimBloco especificam a dimensão do código em blocos e a dimensão dos blocos em *threads*
- Em Fortran para transferir dados entre os dois processadores é apenas necessário igualar as variáveis uma com a outra, por exemplo, x= x_d. Onde o valor da variável x_d guardada na memória do GPU será transferida para a variável x que está alocada na memória do CPU

8.3.1 *Estrutura do código*

Todos os códigos realizados com CUDA devem seguir os passos seguintes:

1. Inicializar e selecionar o GPU em que vai ser executado o código. Por norma o programa irá ser executado no NVIDIA device zero
2. Declarar e alocar dos dados no espaço de memória do GPU
3. Realizar a movimentação dos dados do CPU para o GPU, ou em algum dos casos inicializar a memória no GPU
4. Iniciar *kernels* do CPU para ser executado no GPU
5. Recolher os dados de volta do GPU para análises posteriores ou para ser gerado como *output*
6. Libertar os dados do GPU, que foram alocados no passo 2

Um exemplo simples com todos os passos referidos pode ser o seguinte:

```
program example
    use cudafor; use mytests
    integer, parameter :: n = 100
    integer, allocatable, device :: iarr(:)
    integer h(n)
    istat = cudaSetDevice(0)
    allocate(iarr(n));h = 0; iarr = h
    call test1 <<< 1,n >>> (iarr)
    h = iarr
    write(*,*) h
    deallocate(iarr)
end program example
```

Listing 8.1: Host Code - Exemplo

```
1
module mytests
3 contains
    attributes(global) subroutine test1( a )
5
        integer, device :: a(*)
7       i = threadIdx%x
        a(i) = i
9       return

11  end subroutine test1
end module mytests
```

Listing 8.2: Device Code - Exemplo

Este exemplo foi dividido em dois códigos, sendo o código *host* correspondente ao código que será executado no CPU e o código *host* correspondente ao código que será executado no GPU. Neste exemplo na linha 7 do código *host* o dispositivo GPU é explicitamente selecionado. Na linha 2 deste mesmo código é declarado o módulo *cudafor*, este módulo contém todas as bibliotecas associadas ao CUDA. O array *iarr* é alocado na linha 8. Na linha 9 é realizada uma inicialização dos valores dos dados no CPU e no GPU, e na linha 10 um *kernel* do *host* é invocado. Na linha 11 é realizada a transferência de dados do *host* para o *host*. Por fim na linha 13 é libertada a variável do espaço de memória do *host*. O código do *host* serve para ter uma melhor ideia de como este é constituído e estruturado.

8.3.2 *CUDA Kernels*

CUDA Fortran permite que a definição de sub-rotinas em Fortran seja executada em paralelo no GPU, após serem invocadas pelo programa executado. Estas sub-rotinas são denominadas de *kernels*.

A invocação de um *kernel* específica quantas instâncias em paralelo do *kernel* é que irão ser executadas em paralelo. Cada uma destas instâncias serão executadas em diferentes CUDA *threads*. AS CUDA *threads* estão organizadas dentro de CUDA *blocks*. Cada *thread* tem a informação do bloco a que pertence.

Um *kernel* é identificado utilizando o especificador *attributes global* na declaração da sub-rotina. O que diferencia a invocação de uma *kernel* entre a invocação de uma sub-rotina do CPU é a sua sintaxe. Tendo como exemplo o código seguinte:

```
1
! Kernel definition
3     attributes(global) subroutine exemplo( n, a, x, y )
          real, dimension(*) :: x,y
5         real, value :: a
          integer, value :: n, i
7         i = (blockidx%x-1) * blockdim%x + threadidx%x
          if( i <= n ) y(i) = a * x(i) + y(i)
9     end subroutine exemplo

11 ! Host subroutine
      subroutine exemp( n, a, x, y )
13        real, device, dimension(*) :: x, y
          real :: a
15        integer :: n
          ! call the kernel
17        call exemplo <<< n/64, 64 >>> ( n, a, x, y )
      end subroutine exemp
```

Listing 8.3: Cuda Code - Exemplo 2

Neste caso, na invocação do *kernel* "exemplo"é especificada que serão utilizados $n/64$ *thread blocks*, e que cada bloco terá 64 *threads*. A cada *thread* será atribuída uma variável *blockidx* que corresponde ao número do bloco que esta está associada, e uma variável *threadidx* que corresponde à sua posição dentro do bloco. Neste exemplo, cada *thread* realiza uma iteração do *kernel*.

8.3.3 *Threads, Blocos e Grelhas*

Nesta subsecção irão ser representados e explicados alguns exemplos de como executar um *kernel* com diferentes números de *threads* e *thread blocks*.

Atualmente, o número máximo de *threads* num bloco é de 1024. Como exemplificado nos *listings* 8.1 e 8.3 a invocação de um *kernel* é realizada com uma sintaxe especial. Com esta sintaxe são especificados o número de *threads* e o número de *thread blocks*. Se tivermos algo do tipo : call seq<<<$4,3$>>>(x) as *threads* irão estar organizadas como na figura 35. Sendo a dimensão da grelha em x de 4 blocos e a dimensão dos blocos em x de 3 *threads*. Na figura as *threads* estão numeradas sequencialmente sendo a terceira *thread* do quarto bloco a número 12, mas essa *thread* só tem a informação que o seu *index* em *x* é 3 no bloco de *index* em *x* igual a 4. Logo no *kernel* é necessário efetuar um cálculo para se obter o valor sequencial pretendido. Sendo o cálculo efetuado o seguinte :

$$index\_seq = threadIdx\%x + (blockIdx\%x - 1) * blockDim\%x$$

Aplicado por exemplo na *thread* número 8 temos que:

$$index\_seq = 2 + (3 - 1) * 3 = 8$$

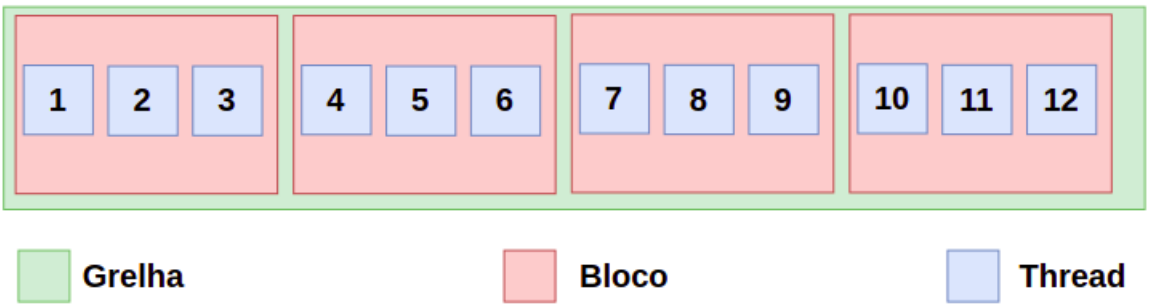


Figura 35: Representação quatro blocos com 3 *threads* cada

Utilizando este *kernel* é possível fazer cálculos em *arrays* de dimensão 12, como exemplificado no *listing* 8.4, em que o *kernel* é uma sub-rotina que irá somar três a todos os valores do *array a*.

```
! Kernel definition
    attributes(global) subroutine exemplo(a, n )
        integer , intent(out):: a(12)
        integer ,value , intent(in):: n
        integer :: i
        i = (blockidx%x-1) * blockdim%x + threadidx%x

        if( i <= 12 ) then
            a(i) = a(i) + n
        end if
    end subroutine exemplo

! Host code
program incTest
    use cudafor
    implicit none
        integer, allocatable :: a(:)
        integer, allocatable, device :: ad(:)
        integer :: n = 3
        allocate (a(12),ad(12))
        a = 5
        ad = a
        call exemplo <<< 4, 3 >>> (ad, n)
        a = ad
        do n = 1, 12
            write(*,*) a(n)
```

```
28      end do

30      deallocate (a,ad)
end program incTest
```

Listing 8.4: Cuda Code - Exemplo 3

O diagrama da figura 35 tem apenas uma dimensão, ou seja as *threads* num bloco e os blocos numa grelha apenas progridem numa dimensão, sendo esta a dimensão de *x*. Este método de utilizar apenas uma dimensão é bastante útil para a utilização de apenas uma dimensão, mas a maior parte dos problemas matemáticos envolvem matrizes. Para resolver com maior facilidade esses problemas CUDA suporta que os blocos de *threads* sejam organizados em uma, duas ou até 3 dimensões em grelhas de blocos, e que as *threads* sejam organizadas também em até 3 dimensões dentro dos blocos. Como discutido nos exemplos anteriores, o primeiro elemento dentro dos <<<>>> é o tamanho da grelha de blocos a ser utilizada e o segundo elemento é correspondente ao tamanho dos blocos de *threads*, nestes exemplos foram apenas utilizados valores de uma dimensão logo a dimensão da grelha e dos blocos também era uni-dimensional. O tipo *dim3*, definido no módulo *cudafor*, pode ser utilizado para declarar varáveis no código do *host* com mais de uma dimensão para ser utilizadas na invocação do *kernel*. Tomemos o código seguinte como exemplo :

```
2 ! Kernel definition
module simpleOps_m
4 contains
  attributes(global) subroutine inc(a,n)
6   implicit none
    integer , intent(out)        :: a(:,:)
8   integer , value , intent(in) ::  n
    integer                      :: x,y
10
    x = (blockIdx%x -1) * blockDim%x + threadIdx%x
12  y = (blockIdx%y -1) * blockDim%y + threadIdx%y

14  if (x < 24 .and. y < 24) then
      a(x,y) = a(x,y) + n
16  end if

18 end subroutine inc
end module simpleOps_m
20
! Host code
22 program incTest
  use cudafor
```

```
24   use simpleOps_m
     implicit none
26   integer :: b, n = 6
         integer, allocatable :: a(:,:)
28       integer, allocatable, device :: ad(:,:)
         integer :: i, k
30       type (dim3) :: dimGrid, dimBlock

32       allocate (a(n,n),ad(n,n))


34
       dimGrid = dim3(3,3,1)
36     dimBlock = dim3(2, 2,1)

38     a= 5
       ad=a
40
           call inc <<< dimGrid, dimBlock >>> (ad, n)
42         a = ad
       do k = 1, n
44       do i = 1, n
           write(*,*) a(n,n)
46       end do
       end do
48
       deallocate (a,ad)
50 end program incTest
```

Listing 8.5: Cuda Code - Exemplo 4

No exemplo do código 8.5 na linha 30 temos a declaração das variáveis *dimGrid* e *dimBlock* como tipo *dim3* e na linha 35 e 36 temos a atribuição dos valores às mesmas. Na linha 41 é invocado o *kernel* com as variáveis *dimGrid* e *dimBlock*, com o uso destas variáveis as *threads* serão organizadas como na figura 36. Onde em *dimGrid* = $dim3(3,3,1)$, estes valores significam que a grelha será constituída por três blocos no eixo x e três blocos no eixo y. Em *dimBlock* = $dim3(2,2,1)$, estes valores representam a organização das *threads* dentro dos blocos, ou seja duas *threads* no eixo x e duas *threads* no eixo y. Tal como no exemplo uni-dimensional as *threads* têm informação sobre a sua posição dentro do bloco, do tamanho do bloco, da posição do bloco na grelha e do tamanho da grelha. Todas estas posições têm coordenadas x e y. Nas linhas 11 e 12 do código 8.5 são feitos os cálculos para obter as coordenadas das *threads* com os valores apresentados na figura 36. Na linha 14 tem uma condição para que os valores não excedam os limites da matriz, e por fim na linha 15 é realizada a incrementação nos valores da matriz.

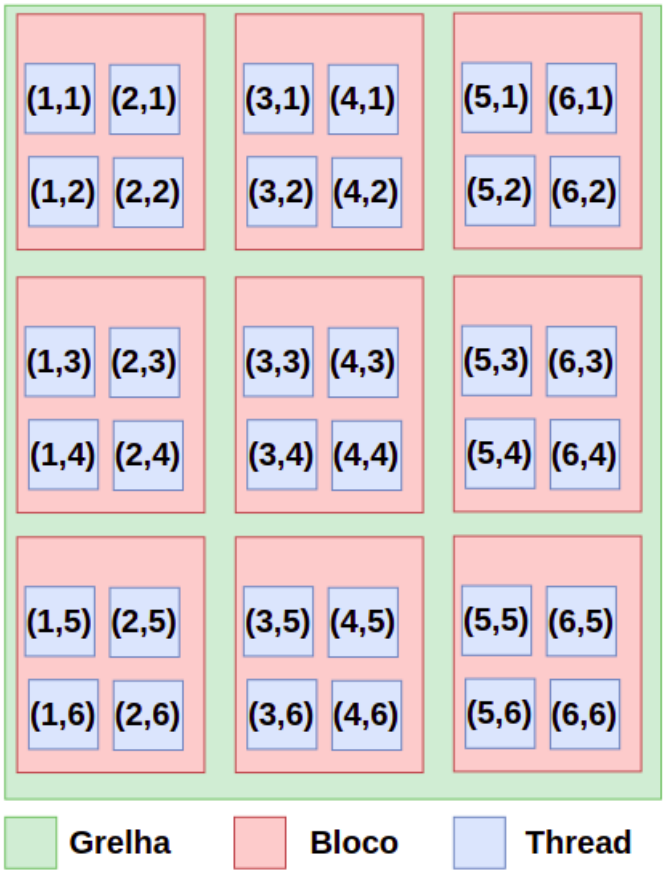


Figura 36: Representação bi-dimensinal da composição de *threads*

8.4 Código com CUDA Fortran

O objetivo de reescrever a secção três do código com CUDA é a de paralelizar todas as subdivisões para que dentro do ciclo não exista a necessidade de transferências de memória entre o CPU e o GPU. Nesta secção irão ser paralelizadas todas as subdivisões da secção três do código e irá ser estudado o seu impacto no tempo de execução desta secção. Inicialmente será realizada a paralelização das subdivisões já paralelizadas com o recurso a OpenACC, em seguida as subdivisões 12 e 13 correspondentes a *shift* de matrizes tanto na horizontal como na diagonal, depois serão reestruturadas as subdivisões 3,4,5 e 6 para poderem ser paralelizadas com CUDA, e por fim será realizada a paralelização das subdivisões 10,17 e 21 com recurso à biblioteca de CUDA denominada de cufft.

8.4.1 *CUDA- Subdivisões de cálculo*

As subdivisões de cálculo são a 2, 7, 8, 9, 11, 14, 15, 16, 18, 19, 20, 22 e 23. Todas estas subdivisões de cálculo, como já discutido, têm a estrutura do tipo:

```
1
do i= 1 , 4096
3     do n = 1, 4096
          C(i,n) = A(i,n) + B(i,n) + D
5     end do
end do
```

Listing 8.6: Estrutura do código das subdivisões de cálculo

É necessário percorrer toda a matriz e efetuar o cálculo em todos os pontos da mesma. Como as dimensões da matriz são de 4096 colunas por 4096 linhas serão necessárias 16777216 operações para executar todos os pontos da matriz, ou cada *thread* realizar o cálculo de mais do que um ponto de cada matriz. Inicialmente será gerada uma *thread* para cada ponto da matriz ou seja 16777216 elementos. A primeira solução criada é do tipo do exemplo seguinte :

```
! Kernel definition
2 module simpleOps_m
contains
4   attributes(global) subroutine inc(A,B,C,D)
  implicit none
6     integer , intent(out)        :: A(:,:)
      integer , intent(out)        :: B(:,:) , C(:,:)
8     integer , value , intent(in) ::  D
      integer                      :: x,y
10
      x = (blockIdx%x -1) * blockDim%x + threadIdx%x
12    y = (blockIdx%y -1) * blockDim%y + threadIdx%y

14    if (x < 4096 .and. y < 4096) then
        A(x,y) = B(x,y) + C(x,y) + D
16    end if

18  end subroutine inc
end module simpleOps_m
20
! Host code
22 program incTest
  use cudafor
24  use simpleOps_m
  implicit none
26      integer :: D, n = 4096
        integer , allocatable , device :: A_dv(:,:), B_dv(:,:), C_dv(:,:)
28      integer , allocatable :: A(:,:)
        integer :: i, k
```

```
30        type (dim3) :: dimGrid, dimBlock

32        allocate (A_dv(n,n),B_dv(n,n),C_dv(n,n))
          allocate (A(n,n),B(n,n),C(n,n))
34
            dimGrid = dim3(128,128,1)
36          dimBlock = dim3(32, 32,1)

38          call inc <<< dimGrid, dimBlock >>> (A_dv,B_dv,C_dv, D)

40          A = A_dv

42          do k = 1, n
              do i = 1, n
44              write(*,*) A(n,n)
              end do
46          end do

48          deallocate (A_dv(n,n),B_dv(n,n),C_dv(n,n))
            deallocate (A(n,n),B(n,n),C(n,n))
50  end program incTest
```

Listing 8.7: Estrutura do código das subdivisões de cálculo

Esta solução para reescrever as subdivisões de cálculo em CUDA utiliza o número máximo de *threads* num bloco, sendo este 1024 = 32 * 32, tendo uma estrutura da organização das *threads* igual ao representado na figura 37. Cada bloco é constituído por 32 linhas e 32 colunas de *threads* e a grelha é constituída por 128 linhas e 128 colunas de blocos, constituindo assim no total 4096 linhas e 4096 colunas de *threads*. Este método é igual ao exemplificado na figura 36 mas com um maior número de *threads* por bloco e um maior número de blocos por grelha.

Com este método temos então as subdivisões de cálculo preparadas para ser paralelizadas e executadas no GPU com CUDA.

Figura 37: Representação de *threads* para matriz de 4096 colunas por 4096 linhas

8.4.2 *CUDA - Subdivisões de Shift*

As subdivisões 12 e 13 têm como função realizar um *shift* circular dos elementos da matriz, sendo que a subdivisão 12 faz horizontalmente e a divisão 13 faz este *shift* verticalmente. Na parte esquerda da figura 38 está representada uma matriz de três colunas por três linhas, em que as setas representam a movimentação de um elemento no eixo horizontal da matriz e a parte direita o resultado final da matriz com o *shift*, sendo este conceito igual ao da subdivisão 12. A figura 39 representa o conceito da subdivisão 13. Na figura 40 está representado o *shift* horizontal e vertical realizado em apenas uma instrução, este conceito será o utilizado na reprogramação das subdivisões 12 e 13 em CUDA, passando estas duas instruções a serem só uma com CUDA.

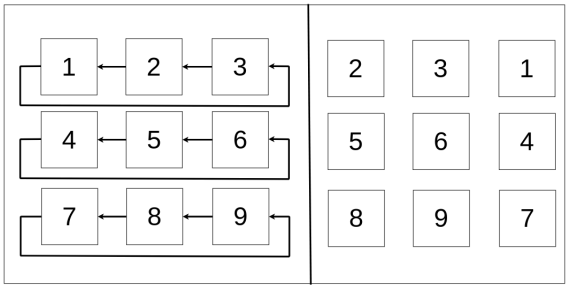


Figura 38: Shift Horizontal de uma matriz

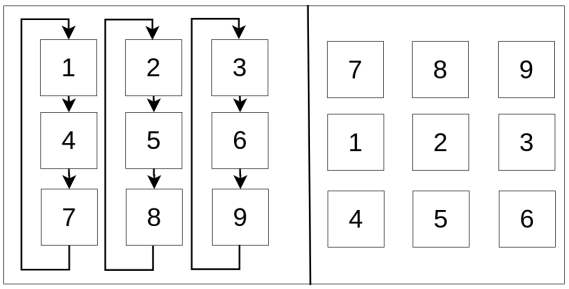


Figura 39: Shift Vertical de uma matriz

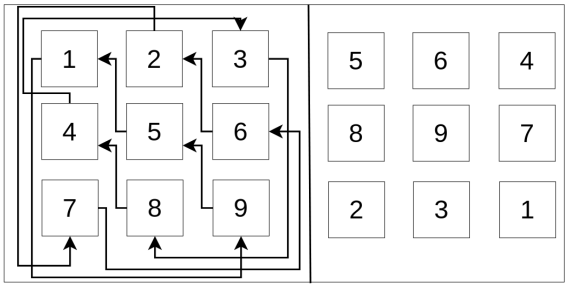


Figura 40: Shift Vertical e Horizontal de uma matriz

Nos três exemplos apresentados anteriormente com o apoio das figuras 38, 39 e 40 a movimentação realizada foi de apenas 1 elemento, sendo que no código a movimentação será de 2048 elementos horizontalmente e verticalmente.

A função de *shift* foi realizada com a mesma estrutura do exemplo seguinte:

```
attributes(global) subroutine shift_dv(A,B)

    implicit none

    complex, intent(out)  :: A(:,:)
    complex , intent(in)  :: B(:,:)
    integer :: x,y
```

```
    x = (blockIdx%x -1) * blockDim%x + threadIdx%x
    y = (blockIdx%y -1) * blockDim%y + threadIdx%y

    if (x > 2048 .and. y > 2048) then

      A(x,y) = B(x-2048,y-2048)

    else if ( x < 2049 .and. y > 2048) then

      A(x,y) = B((4096-(2048-x)),y-2048)

    else if ( y < 2049 .and. x > 2048) then

      A(x,y) = B(x-2048,(4096-(2048-y)))

    else
      A(x,y) = B((4096-(2048-x)),(4096-(2048-y)))

    end if

end subroutine shift_dv
```

Listing 8.8: Estrutura do código das subdivisões de cálculo

Neste exemplo o *shift* é realizado na matriz *a*, antes do kernel shift_dv ser invocado todos os valores da matriz *a* foram passados para a matriz *B* para que estes pudessem ser usados na função de *shift* criada. O organização das *threads* utilizadas para percorrer toda a matriz de modo a efetuar o *shift* é igual ao da figura 37

8.4.3 *CUDA - Subdivisões de procura*

As subdivisões 3 e 4 são idênticas, sendo que a três percorre um *array* para encontrar o valor mais próximo do valor mínimo de uma matriz e a quatro percorre o mesmo para encontrar o valor mais próximo do valor máximo dessa matriz. Ambas as subdivisões no ciclo são uma invocação de uma função com cinco argumentos, sendo eles: o primeiro uma variável com o tamanho do segundo argumento que é um *array* uni-dimensional, o terceiro elemento é o valor mínimo real da variável denotada por um, na figura 26, e o quarto e quinto argumentos são variáveis em que serão guardados dois valores. Para melhor compreensão da subdivisão segue o exemplo seguinte:

```

    integer          :: Rpts, i ,imin, imax
    double precision :: variavel_4(:,3)
    real             :: variavel_1(4096,4096)
```

```
5
    ! inicializacoes de variaveis e outros cálculos
7                               .
    Rpts = size(variavel_4,1)
9                               .

11   call subdivisao_3 (Rpts, variavel_4(:,1), real(minval(variavel_1)),
    imin, i)

13   call subdivisao_4 (Rpts, variavel_4(:,1), real(minval(variavel_1)),
    i, imax)
```

Listing 8.9: Estrutura do código das subdivisões de cálculo

O objetivo desta subdivisão é encontrar os dois valores mais próximos do *array variavel_4* em relação ao valor mínimo da matriz *variavel_1* no caso da subdivisão três e máximo no caso da subdivisão quatro. Com o apoio da figura 42 será explicado o método utilizado para encontrar estes 2 valores. Neste exemplo o *array variavel_4* seria de tamanho 12, logo o valor de *Rpts* seria doze. O valor a ser procurado será de 6.5, no caso da função *subdivisao_3* do código este valor corresponderia a *real(minval(variavel_1))*. Na primeira iteração o valor de *imin* é colocado na posição um e o de *i* na posição doze representado na imagem a azul, e o cálculo a ser efetuado é o de:

$$int(\frac{imin + i}{2}) = 6$$

Como 6.5 >6 o valor de *i* mantém-se na mesma posição e o valor de *imin* irá ser igual ao valor obtido no cálculo, ou seja 6. Na segunda iteração é feito o mesmo procedimento ou seja :

$$int(\frac{imin + i}{2}) = 9$$

Como 6.5 <9 o valor de *imin* vai manter-se na mesma posição e o valor de *i* será igual ao valor obtido no cálculo. Isto será repetido até que o valor de *i* - *imin* seja maior que um. Na quarta iteração temos que 7-6=1, sendo um não maior que um, esta é a iteração final. Com a utilização deste método apenas é necessário realizar quatro iterações até encontrar os valores desejados. Se a procura fosse feita por comparar todos os valores do *array* com os valores pretendidos seriam necessárias doze iterações ao invés de quatro. Apesar de este método ser bastante eficaz todas as iterações têm de ser sequências eliminando assim a possibilidade da paralelização do mesmo. Sendo estas duas subdivisões as únicas que não necessitam de ser sequências ambas serão executadas em paralelo. Atribuindo assim uma *thread* para cada subdivisão.

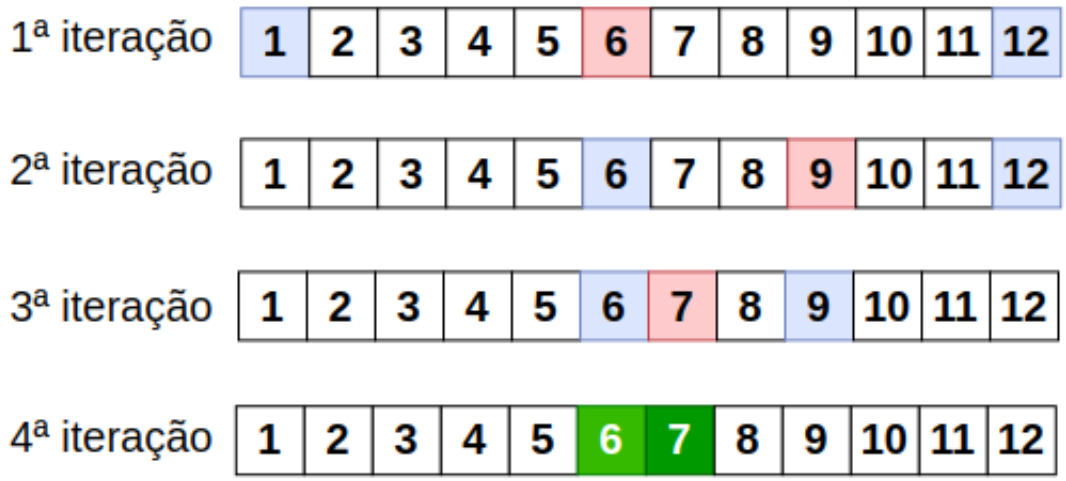


Figura 41: Método de pesquisa de valores num array

O código desenvolvido em CUDA das subdivisões 3 e 4 em CUDA foi o seguinte:

```
!Host Code

    call subdivisao_2.....

    variavel_1 =  variavel_1_dv

    call subdivisao_3_4 <<< 1, 2 >>> (Rpts, variavel_4(:,1), real(
   minval(variavel_1)), real(maxval(variavel_1)), imin, imax)


!Device code

!declaracao de variaveis ...

x = threadIdx%x

  if (x ==1 ) then
      r_1 = Rpts
      l_1 = 1

    do while (r_1-l_1 > 1)

      m_1 = int( (l_1+r_1)/2 )

      if ( variavel_4 (m_1) > xmin ) then
        r_1 = m_1
```

```
      else
28       l_1 = m_1
      end if
30
    end do
32
    left = l_1
34
  end if
36                    .
                      .
38                          .
```

Listing 8.10: Estrutura do código das subdivisões de procura

Com este código existe a necessidade de transferência do valor de variavel_1 do GPU para o CPU, porque na invocação do *kernel* da linha 8 e 9 este valor é utilizado pelo CPU para realizar a função de *minval* e *maxval*. Após a linha 15 temos o código que será executado no GPU, sendo este referente à primeira *thread* que irá realizar a procura do valor mais próximo do valor mínimo da variável um.

8.4.4 *Interpolação*

Na subdivisão cinco é realizada uma interpolação linear. Nesta subdivisão são realizados cálculos em 33554432 elementos de um *array* de três dimensões. Sendo a primeira e segunda dimensões deste *array* constituídas por 4096 colunas e 4096 linhas e a terceira dimensão constituída por apenas 2 elementos. O eixo Z, constituído por dois elementos é necessário para a formação de números complexos na subdivisão seguinte. Na subdivisão cinco antes do cálculo é também realizada uma procura de valores utilizando o método da figura 42.

Para a execução desta função com CUDA a organização inicial das *threads* será igual à representada na figura 37, em que cada *thread* realizará a procura do valor com o método discutido num *array* de tamanho 16777216 e efectuará dois cálculos sendo um para Z=1 e outro para Z=2.

8.4.5 *Transformadas de Fourier*

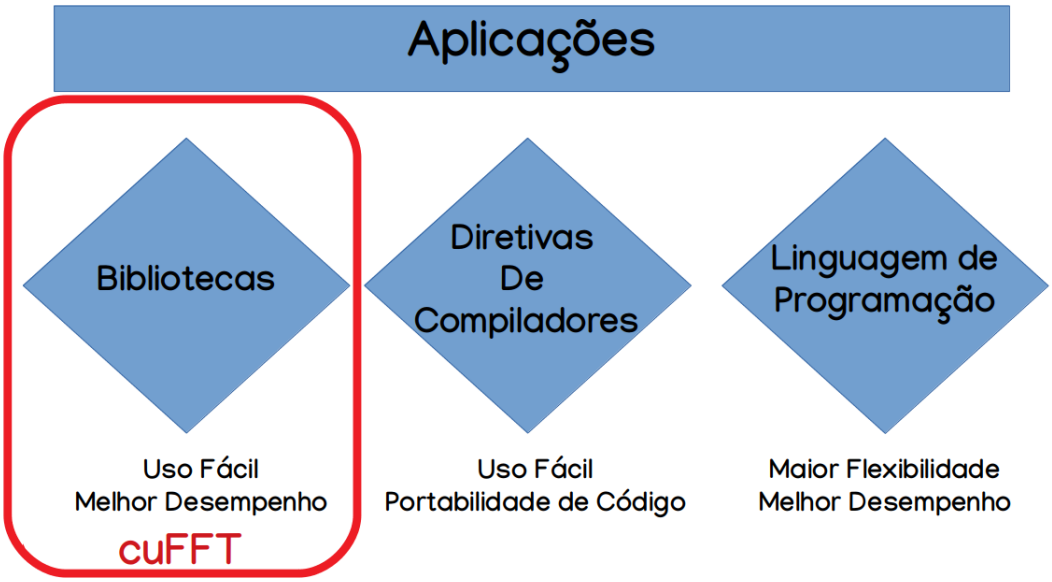


Figura 42: Forma de otimização - Bibliotecas

A Transformada de Fourier de uma sequência complexa é descrita da seguinte forma:

$$X(k) = \sum_{j=0}^{N-1} x[n] e^{-jk\frac{2\pi}{N}n}; \qquad k = 0, 1, ..., N-1 \tag{3}$$

A diferença principal entre computação de FFT no GPU e no CPU é que o valor de n é gerado como uma função do número da *thread* t, do bloco b e do número de *threads* por bloco T.

A biblioteca cuFFT da NVIDIA fornece um interface simples e eficaz para a computação de FFTs em GPUs da NVIDIA. Como discutido na secção 5 uma das formas de retirar o maior proveito de um dispositivo é utilizar as bibliotecas fornecidas pelo fornecedor do mesmo. Logo para a realização das subdivisões de cálculo das FFT será utilizada a biblioteca cuFFT da NVIDIA.

8.5 Compilação e tempos de execução

O código desenvolvido na secção anterior foi compilado com o compilador PGI. Todos os *kernels* foram invocados com a disposição de *threads* representadas na figura 37, excepto os *kernels* das subdivisões de procura e das subdivisões das transformadas de fourier. Os *kernel* das subdivisões de procura é invocado com duas *threads* em apenas um bloco e nas subdivisões das transformadas de fourier é realizada a invocação à função "cufftExecC2C" da biblioteca cuFFT.

Os tempos de execução de cada subdivisão estão representados na tabela 11. Na tabela a subdivisão quatro tem tempo de 0 segundos porque houve uma fusão das subdivisões três e quatro, estando

o tempo desta fusão representado na subdivisão três. O mesmo aconteceu com a subdivisão doze e treze, em que o tempo dessa fusão esta representado na subdivisão doze.

Tabela 11: Tempo de cada subdivisão em 100 iterações - CUDA

Subdivisão	Tempo em segundos em 100 iterações	Subdivisão	Tempo em segundos em 100 iterações
1	0,0000725	13	0
2	1,56	14	1,2
3	17,9	15	0,841
4	0	16	0,0000153
5	4,39	17	1.97
6	0,9757	18	0.66
7	13,324	19	1,2
8	6,94	20	1,16
9	1,159	21	1.97
10	1,97	22	0.66
11	0.66	23	1,2
12	1,36		

Sintetizando, com este ensaio foram obtidos os seguintes resultados:

- Tempo total dos 100 ciclos : 61,1 segundos
- As subdivisões 3 e 7 são as que exigem maior poder computacional, sendo que estas ocupam cerca de 51% do tempo de computação do ciclo
- Tempo total de cada ciclo em média : 0,611 segundos
- Tempo total da Divisão 2 : 24,83 segundos
- Tempo total da Divisão 3 : 36,28 segundos

8.5.1 *Comparação - Sequencial vs CUDA paralelo*

Na Fig. 43 estão representados a azul os tempos das subdivisões paralelizadas com a linguagem de programação CUDA, e a laranja as subdivisões que correspondem às subdivisões sequências correspondentes à compilação do código com o compilador PGI. Na totalidade os tempos de execução das 100 iterações com o código sequencial foram de 623 segundos e os do código paralelizado de 61 segundos aproximadamente. O tempo de execução da secção três diminuiu 90% após a paralelização

do mesmo. Todas as subdivisões excepto a subdivisão 3 e 4 obtiveram melhor desempenho após serem paralelizadas. As subdivisões 10, 17 e 21, correspondentes ao cálculo da FFT demoraram aproximadamente 4% com o uso da biblioteca cuFFT da NVIDIA comparativamente com o uso da biblioteca fftw.

São visíveis as melhorias significativas dos tempos de execução do código com a simples paralelização do mesmo para este ser executado em GPU. Com esta paralelização foi possível diminuir os tempos de execução da secção 3 do código de 5.74 segundos por iteração para 0.611 segundos por iteração. Estas são melhorias significativas, pois em apenas 100 iterações foi possível reduzir o tempo da secção em 8 minutos e 32 segundos, sendo esta uma redução de 90%. Com este ensaio já é possível observar a grande potencialidade do GPU para ajudar a resolver problemas relacionados com a Engenharia e a Ciência.

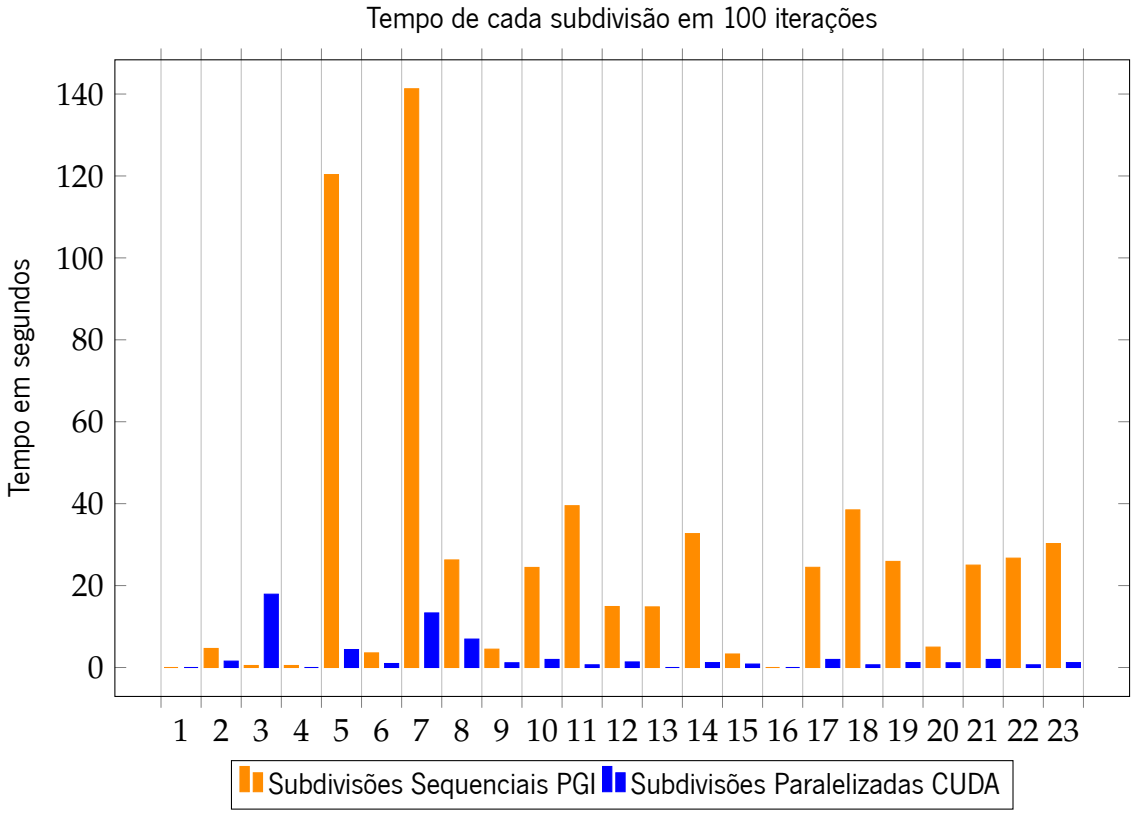


Figura 43: Gráfico de comparação - PGI vs GFortran

8.5.2 *Análise PGI Profiler*

Foi analisada a execução do código com o programa PGI Profiler para obter uma melhor percepção do motivo da subdivisão 3 e 4 obter um desempenho inferior quando paralelizada. Na figura 44 estão representados os dados obtidos com o PGI Profiler para a subdivisão 3 e 4.

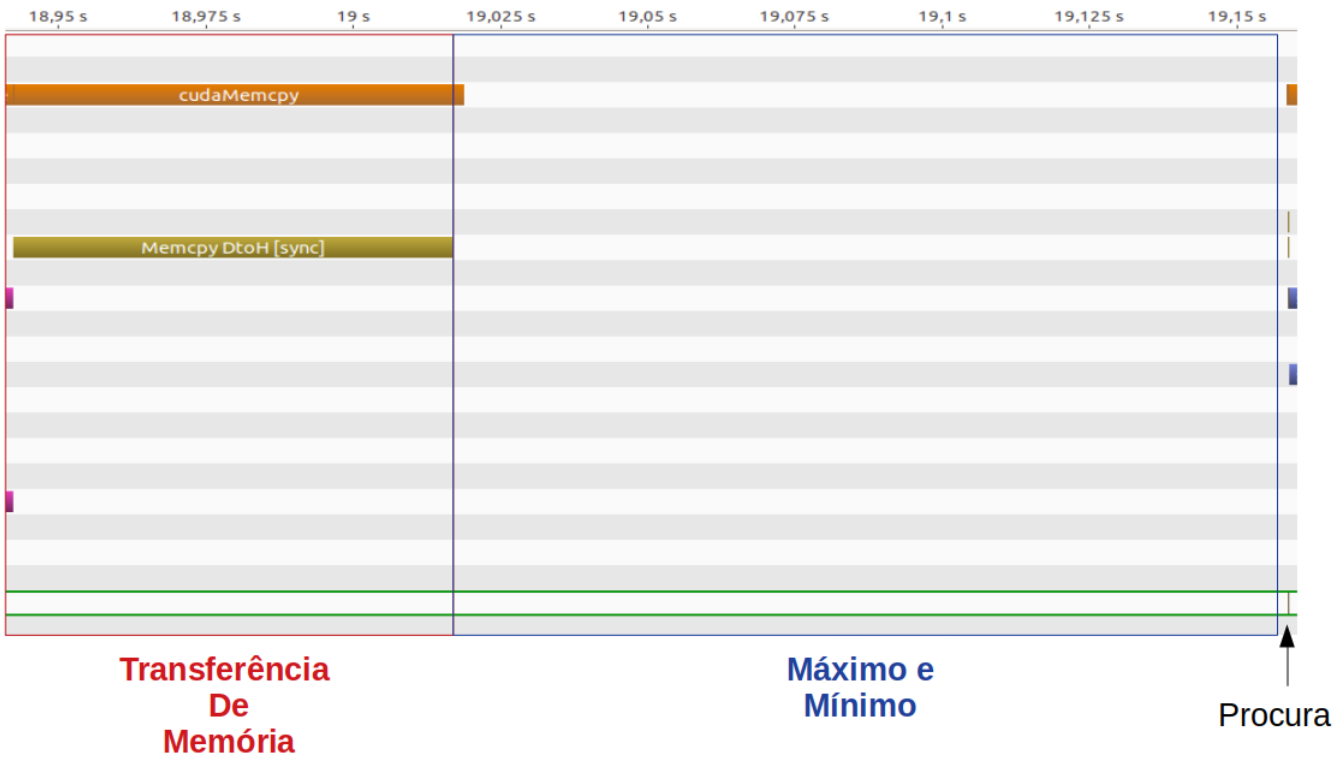


Figura 44: Subdivisões de procura

Com o recurso da análise realizada com o programa PGI Profiler é possível perceber que mais de 55% do tempo de execução da subdivisão 3 e 4 é ocupado pela procura do máximo e mínimo da variável 1, que é realizado pelas funções executadas no CPU (”minval”e ”maxval”) antes da execução do *kernel* do GPU. Devido à necessidade de executar as funções ”minval”e ”maxval”no CPU, existe consequentemente a necessidade de realizar a transferência de memória do valor da variável 1 do GPU para o CPU antes destas funções serem realizadas. O tempo de execução do *kernel* criado é de microssegundos sendo que as duas funções do CPU e a transferência de memória ocupam aproximadamente 178 milissegundos por iteração. Para eliminar a necessidade da transferência de memória é necessário realizar as funções de procura do valor máximo e mínimo da variável 1 através do GPU.

8.6 Otimização direccionada para a máquina

Nesta secção serão efetuadas otimizações ao código inicial escrito em CUDA Fortran. Como já discutido, para obter uma melhor otimização através da utilização de uma linguagem de programação como CUDA é necessário fazer programação direccionada para o GPU em que será executado o programa. Primeiramente será feito o estudo das características do GPU que está a ser utilizado, e em seguida serão analisadas as formas de execução mais adequadas para o mesmo.

8.6.1 *Características do GPU*

O GPU utilizado tem as seguintes características descritas na tabela 12.

Tabela 12: Características do GPU utilizado

Name	GeForce MX150
CUDA Driver Version	10.2
CUDA Capability	6.1
Multiprocessors (SM)	3
Max Thread Blocks per SM	8
Total amount of global memory	2004 Mb
GPU Max Clock rate	1.53 GHz
Memory Clock rate	3 GHz
Memory Bus Width	64 bits
Shared Memory per block	49152 bytes
Total Number of registers available per block	65536
Warp size	32
Maximum number of threads per multiprocessor	2048
Maximum number of threads per block	1024
Max dimension size of a thread block (x,y,z)	(1024,1024,64)

O GPU utilizado possui três multiprocessadores. As limitações do multiprocessador são:

- Só pode executar simultaneamente um máximo de 8 *thread blocks*
- O máximo de *threads* executadas simultaneamente por multiprocessador é de 2048
- O número máximo de *registers* utilizados pelas *threads* de cada multiprocessador é de 65536
- O número máximo de bytes por memória partilhada de cada bloco é de 49152 bytes
- O número máximo de *threads* num *thread block* é de 1024

Warps

O multiprocessador cria, gere, aloca e executa as *threads* em grupos de 32 *threads* paralelas chamadas de *warps*. Todas as *threads* individuais que compões um *warp* começam ao mesmo tempo no programa. Quando é atribuído a um multiprocessador mais do que um *thread block*

para ser executado, esse *thread block* é repartido em *warps*. Uma *warp* executa uma instrução comum de cada vez, logo para obter eficácia máxima todas as *threads* da mesma *warp* devem seguir o mesmo "caminho". Se alguma *thread* divergir, por exemplo devido a uma condição, esta apenas será executada após todas as outras acabarem de ser executadas. Devido às *threads* serem agrupadas em grupos de 32 em cada *thread block*, para otimizar a performance do bloco é normalmente utilizado um número de *threads* múltiplo de 32.

Memory Coalescing

Como já discutido e representado na figura 33 existem 3 tipos de memória que cada *thread* pode utilizar, sendo a memória local apenas respetiva a cada *thread*, a memória partilhada respetiva a cada bloco e a memória golobal respectiva a cada grelha ou *kernel*. O acesso à memória global é realizado em grandes parcelas de 32 a 128 bytes de uma vez por cada *warp*. Imaginemos o *array* 1, este *array* contém doze valores e estes doze valores estão armazenados na memória global do GPU, da forma representada na figura 45.

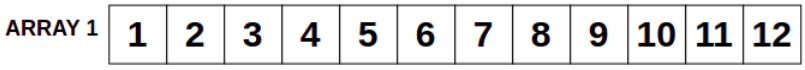


Figura 45: Array 1 armazenado na memória global do GPU

Para este exemplo imaginemos que o número de *threads* por *warp* é de quatro. Se cada *thread* só necessitar de aceder a uma posição de memória e se as quatro *threads* do *warp* acederem a posições de memória consecutiva temos algo parecido com o ilustrado na figura 46. Neste exemplo o *warp* 1 apenas necessita realizar uma transferência de memória, pois a parcela acedida contém toda a informação necessária para a execução de todas as *threads* do *warp*.

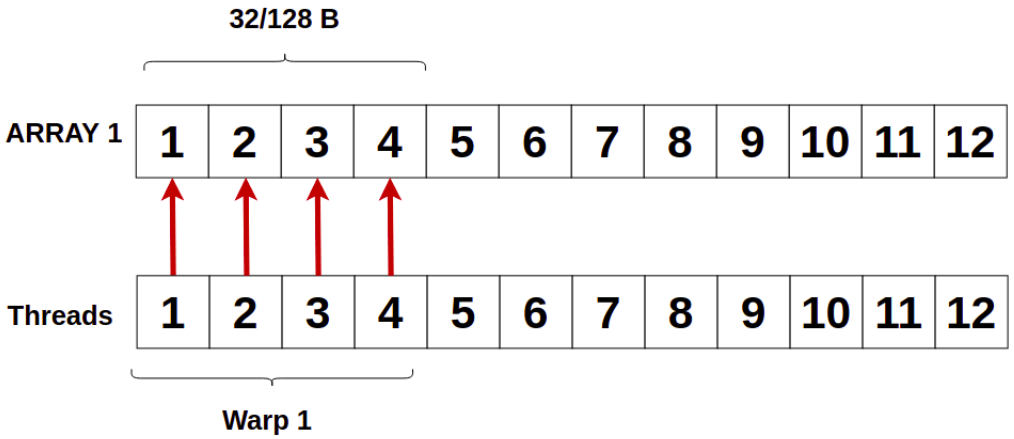


Figura 46: Forma mais eficaz de acessar memória no GPU

Um método ineficaz de acessar memória global é o representado na figura 47, neste exemplo o *warp* 1 necessita de realizar duas transferências de memória e alguns valores do *array* 1 acedidos não iriam ser utilizados. Sendo que com esta ineficácia de acesso de memória iria perder-se mais tempo porque seria necessário realizar duas transferências de memória.

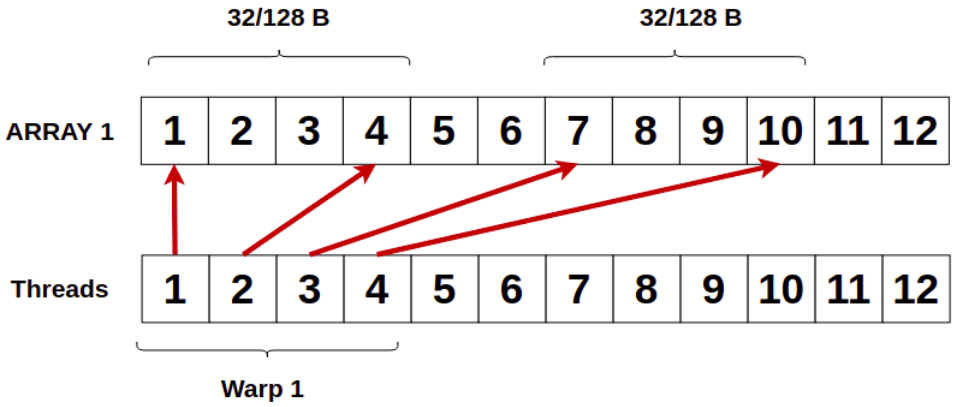


Figura 47: Forma mais ineficaz de aceder memória no GPU

8.6.2 *Otimização Subdivisão 3 e 4*

Grande parte do tempo necessário para a execução da subdivisão 3\4 é devido à necessidade da utilização das funções "maxval"e "minval"do CPU. Estas funções obrigam a que haja uma transferência de memória a ocorrer antes de cada função realizada no CPU. Para realizar a procura do valor máximo e mínimo de uma matriz é necessário realizar a comparação de todos os valores da matriz para depois conseguir obter quais desses valores é o máximo e qual é o mínimo.

O primeiro instinto de um programador que trabalha constantemente com código sequencial seria o de atribuir cada valor da matriz a uma *thread*, e em cada *thread* realizar a comparação do valor armazenado na memória local da *thread* com o valor alocado na memória global do GPU, sendo este o valor máximo ou mínimo. Esta opção é impraticável, apesar de todas as *threads* conseguirem ler o valor da memória global e de todas estas conseguirem modificar este valor , estas podem estar a ler o valor errado. Como já discutido, as *threads* são executadas em *warps* ao mesmo tempo, fazendo com que todas as *threads* do mesmo *warp* obtenham o valor armazenado na memória global aproximadamente ao mesmo tempo. Imaginemos o seguinte caso, as 32 *threads* de um *warp* obtêm o valor máximo de 30 que estava armazenado na memória global, a *thread* 1 tem armazenado na sua memória local o valor 40 e a *thread* 2 tem armazenado na sua memória local o valor de 35, sendo ambos os valores maiores do que o valor máximo já existente. Neste caso é possível acontecer que a *thread* 1 escreva na memória global o valor 40 e em seguida a *thread* 2 modifique este valor para 35, sendo este valor menor do que o valor da *thread* 1. Devido à

possibilidade da ocorrência deste erro, este método é impraticável em qualquer código que seja executado em paralelo.

O método utilizado para resolver este problema foi o seguinte:

1. Foi atribuído a cada memória local de cada *thread* o valor correspondente a uma posição da matriz.

2. Foram declaradas duas matrizes de dimensão 32x32 na memória partilhada de cada *thread block*, uma para o valor mínimo e outra para o valor máximo, foi utilizada a memória partilhada de cada *thread block* ao invés da memória global devido à sua maior largura de banda, fazendo assim com que as operações de transferência de dados sejam mais rápidas.

3. Cada *thread* escreve o seu valor na posição respetiva da matriz de memória partilhada. Por exemplo, se a *thread* tem as coordenadas (5,6) dentro do *thread block*, esta irá escrever o seu valor na posição (5,6) da matriz alocada na memória partilhada.

4. É utilizada uma barreira para sincronizar todas as *threads*, e esta barreira faz com que todas as *threads* só passem para a próxima instrução após todas as *threads* terem chegado a este ponto.

5. É realizada uma redução das matrizes alocadas na memória partilhada, e esta redução serve para encontrar o valor máximo e mínimo presente em cada *thread block*. A redução é dividida em dois, sendo a primeira a redução vertical e a segunda horizontal.

   A primeira parte da redução é realizada da mesma forma que a do exemplo da figura 48. Neste exemplo temos uma matriz de dimensões 8x8 em que a *thread* (1,1) é responsável pelo valor da posição (1,1), a *thread* (1,2) pelo valor da posição (1,2) e assim sucessivamente. A cada iteração apenas as *threads* correspondentes à metade superior da matriz irão realizar a comparação com os valores das *threads* da parte inferior da matriz. Por exemplo, na primeira iteração a *thread* da posição (2,2) irá comparar o valor da matriz (2,2) com o valor da matriz (2,4) e armazenar o maior valor na posição (2,2) no caso da procura do valor máximo. Na segunda iteração apenas existe metade da matriz da iteração anterior, e este processo será realizado sucessivamente até que apenas existe uma linha, no caso da redução vertical.

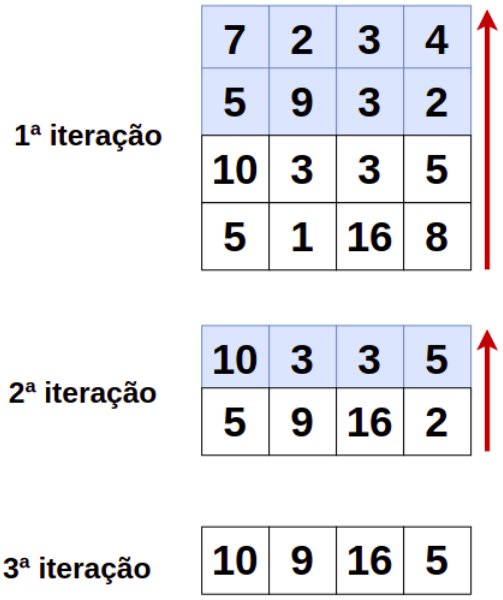


Figura 48: Redução vertical de uma matriz de dimensão 8x8

A segunda parte, sendo esta a redução horizontal, o procedimento é o mesmo só que na vertical. As iterações só acabam quando existir apenas um valor na posição (1,1). Esta redução está representada na figura 49, obtendo assim o valor máximo de cada *thread block*. Na primeira iteração o valor 10 é comparado com o valor 16 e o valor 9 com o 5.

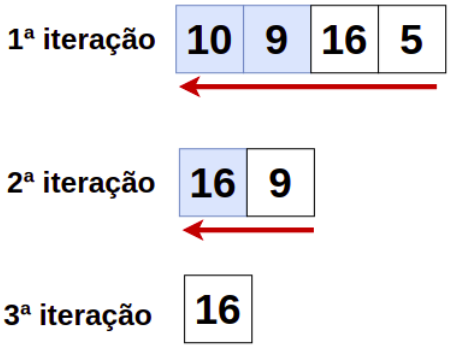


Figura 49: Redução horizontal de uma matriz de dimensão 8x1

6. Após todas os *thread blocks* terem os seus valores máximos e mínimos armazenados na posição (1,1) dos *arrays* alocados na suas memórias partilhadas é realizada uma sincronização das *threads* do bloco

7. Como todos os *thread blocks* têm o seu valor máximo e mínimo armazenado na posição (1,1) dos *arrays* correspondentes cada *thread* de coordenadas (1,1) irá fazer a comparação desse valor com um valor armazenado na memória global do GPU. Para evitar o erro já discutido de uma *thread* ler o valor errado foram utilizadas funções atómicas

As funções atómicas garantem que a operação é desempenhada sem a interferência de nenhuma outra *thread* . Por outras palavras, nenhuma outra *thread* consegue aceder ao endereço de memória a ser alterado até que a operação esteja concluída. As funções atómicas utilizadas foram a "atomiccas"e a "atomicexch". Com estas duas funções é possível realizar uma espécie de fechadura, ou seja, quando a primeira *thread* realizar a comparação do valor máximo global com o seu valor máximo irá fechar a fechadura para que todas as outras *threads* que quiserem fazer esta operação tenham de esperar, e só após esta *thread* realizar a comparação e escrita, se necessário, é que esta *thread* irá abrir a fechadura para outra *thread* realizar a comparação

Análise PGI Profiler

Realizando uma nova análise da subdivisão 3 e 4 obtemos a figura 50. Nesta figura podemos observar que foram eliminados todos os tempos de execução da secção 3 no CPU e que já não existem transferências de memória entre os dois processadores.

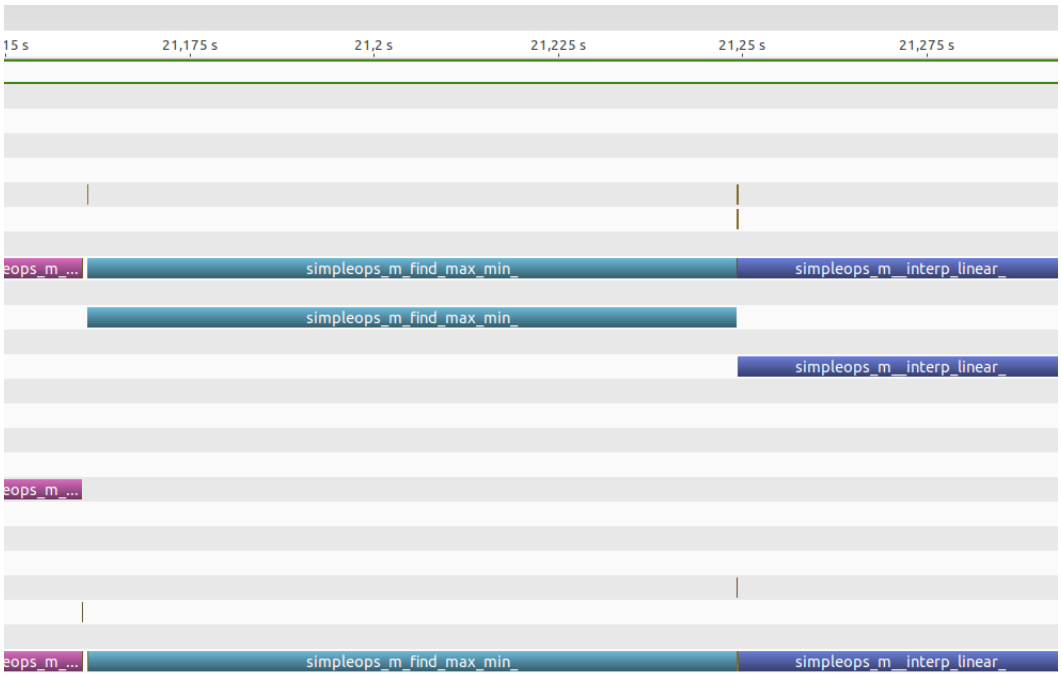


Figura 50: Subdivisões de procura

Com esta otimização, o tempo de execução da subdivisão em 100 iterações foi de 8.738 segundos, sendo neste momento o tempo de execução da secção 3 em 100 iterações de aproximadamente 51.94 segundos.

8.6.3 *Coalesced*

Como discutido anteriormente, é importante para o desempenho de um *kernel* que a memória seja lida de forma *coalesced*. Os resultados até agora apresentados foram todos de *kernels* que realizavam a transferência de dados desta forma. Para uma melhor análise à importância de utilizar o método *coalescing* iremos comparar os resultados de várias subdivisões com e sem este método. Esta comparação está representada na tabela 13. Os tempos de execução com *coalescing* são menores do que sem *coalescing*, este aumento de desempenho é mais notório em *kernels* que o seu tempo de execução é maioritariamente ocupado por transferências de dados. Como as subdivisões 7 e 8 são subdivisões em que o tempo de execução é maioritariamente ocupado por instruções de cálculo não é, tão observável o aumento de desempenho após o uso do método de *coalescing*.

Tabela 13: Com coalescing vs Sem coalescing

Subdivisão	Com coalescing	Sem coalescing	Sem / Com (em segundos)
2	1.56	1.8	2.808
6	0.976	2.037	2.087
7	13.324	13.65	1,024
8	6.94	7.27	1.048
9	1.159	3.27	2.821
11	0.66	1.65	2.5
12	1.36	2.37	1.74
14	1.2	1.81	1.508
15	0.841	2.5	2.97
18	0.66	1.65	2.5
19	1.2	1.81	1.508
20	1.16	3.26	2.81
22	0.66	1.65	2.5
23	1.2	1.81	1.508

Com a utilização de *coalescing* nestas subdivisões o tempo de cada iteração diminuiu 130 milissegundos, o que em 100 iterações representa 13 segundos. Sendo assim, este método de ler e escrever dados uma ótima forma de otimizar cada *kernel*.

8.6.4 *Elementos por thread*

Nesta secção irão ser estudados os desempenhos de cada *thread* no processamento de diferentes números de elementos. Até este ponto cada elemento das matrizes era processado por uma *thread*, ou seja cada *thread* era apenas responsável pelo processamento de um elemento. Com este método está a ser utilizado o máximo paralelismo possível. Com o intuito de estudar se utilizar o método de máximo paralelismo é o mais eficaz para todas as subdivisões, foram efetuados vários ensaios onde em cada ensaio cada *thread* processa um número de elementos diferente.

No gráfico da figura 51 temos representadas a evolução do tempo de execução das subdivisões 2, 6, 9 e 15 com o aumento de elementos a serem processadores por cada *thread*. Com este gráfico é possível observar uma diminuição de aproximadamente 0.23 segundos em 100 iterações com o processamento de 32 elementos por *thread* na subdivisão 2. Nas subdivisões 6, 9 e 15 a diferença de tempos de execução por número de elementos por *thread* não apresenta uma grande variação.

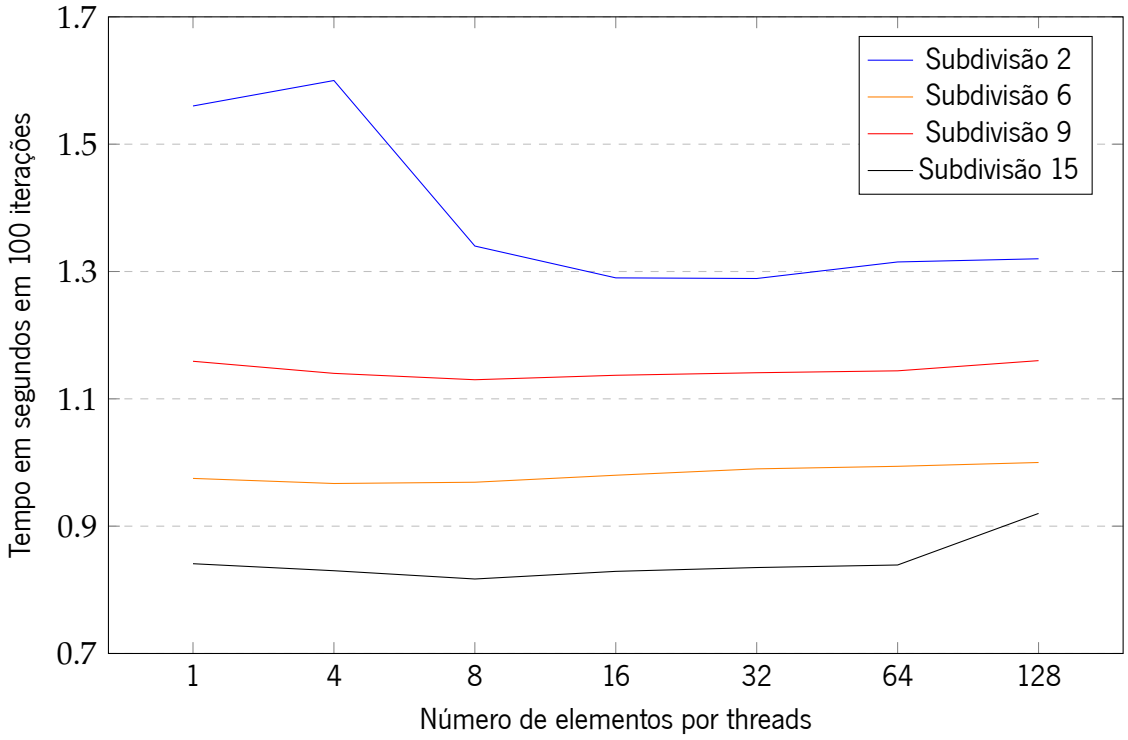


Figura 51: Gráfico da evolução do tempo de execução em relação ao aumento de elementos por thread no GPU - Subdivisões 2,6,9 e 15

No gráfico da figura 52 temos representadas a evolução do tempo de execução das subdivisões 5, 7 e 8 com o aumento de elementos a serem processadores por cada *thread*. Com este gráfico é possível observar uma diminuição de aproximadamente 5.25 segundos em 100 iterações com o processamento de 32 elementos por *thread* na subdivisão 7. Na subdivisão 8 existe uma diminuição de 2.1 em 100 iterações com o processamento de 32 elementos por *thread* comparativamente com o processamento de apenas um elemento por *thread*. A subdivisão 5 apresenta o seu melhor resultado com apenas um elemento por *thread*.

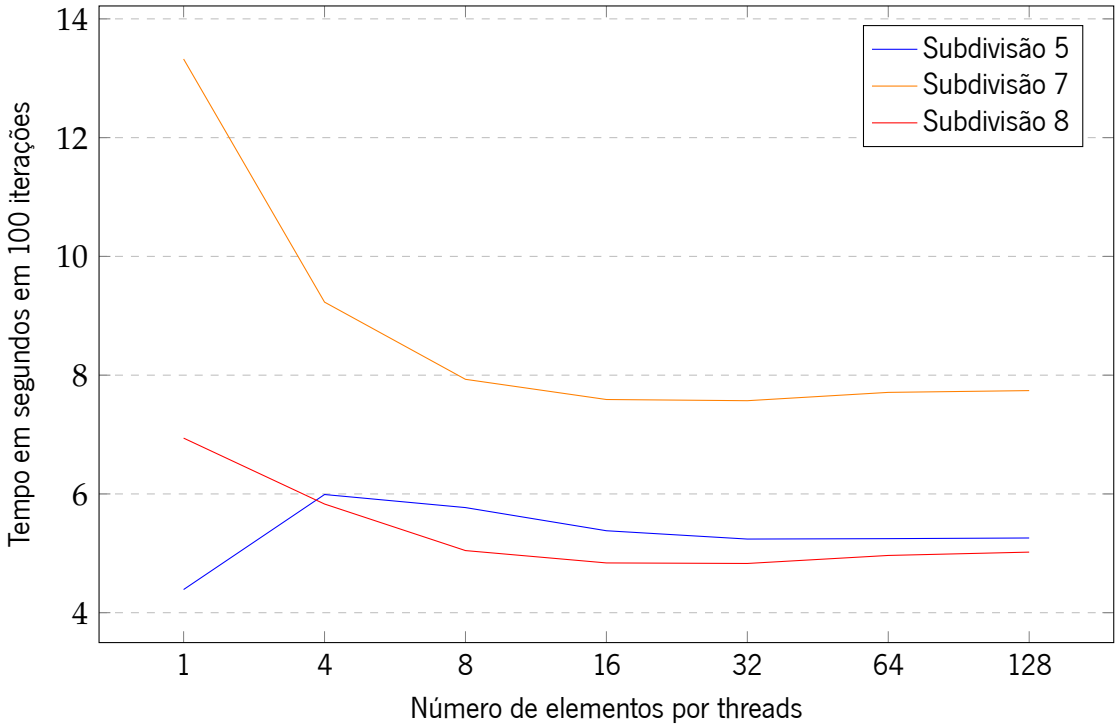


Figura 52: Gráfico da evolução do tempo de execução em relação ao aumento de elementos por thread no GPU - Subdivisões 5,7 e 8

No gráfico da figura 53 temos representadas a evolução do tempo de execução das subdivisões 11, 18 e 22 com o aumento de elementos a serem processados por cada *thread*. Este gráfico ajuda a perceber que nas subdivisões 11, 18 e 22 quanto maior o número de elementos por *thread* pior a performance da subdivisão.

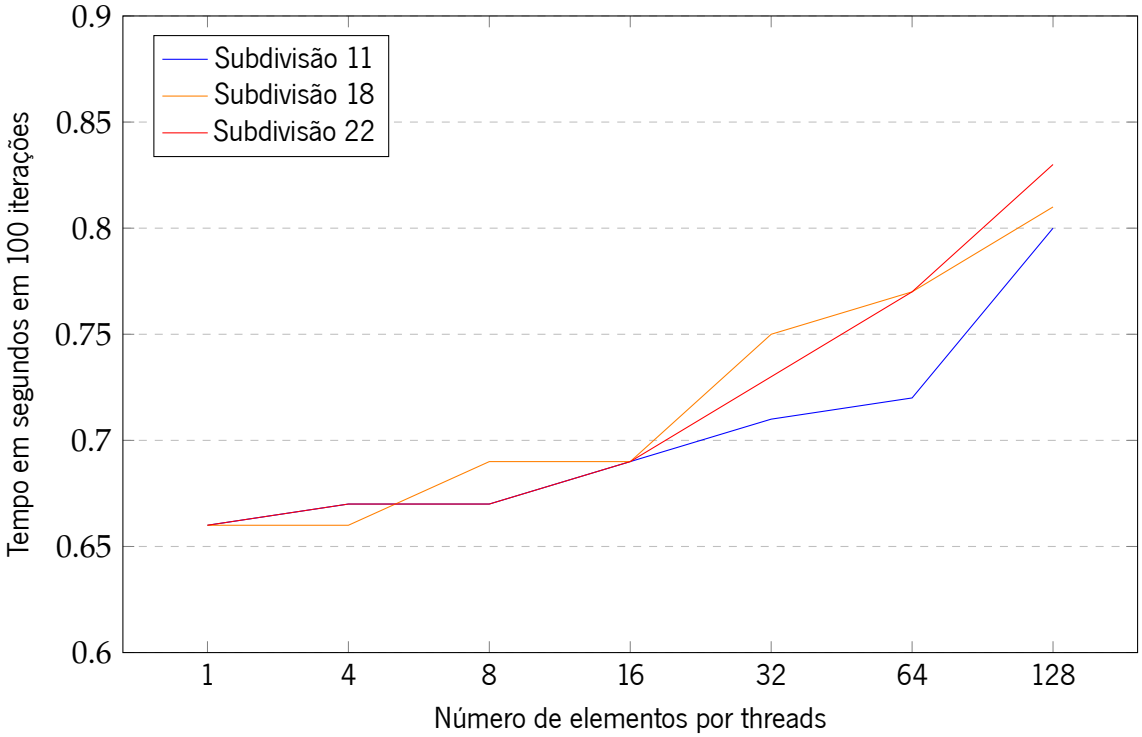


Figura 53: Gráfico da evolução do tempo de execução em relação ao aumento de elementos por thread no GPU - Subdivisões 11,18 e 22

No gráfico da figura 54 temos representadas a evolução do tempo de execução das subdivisões 14, 19, 20 e 23 com o aumento de elementos a serem processados por cada *thread*. As subdivisões 14, 19 e 23 obtêm o seu pico de desempenho com 32 elementos por *thread*, já a subdivisão 20 obtém o seu pico com 8 elementos por *thread*.

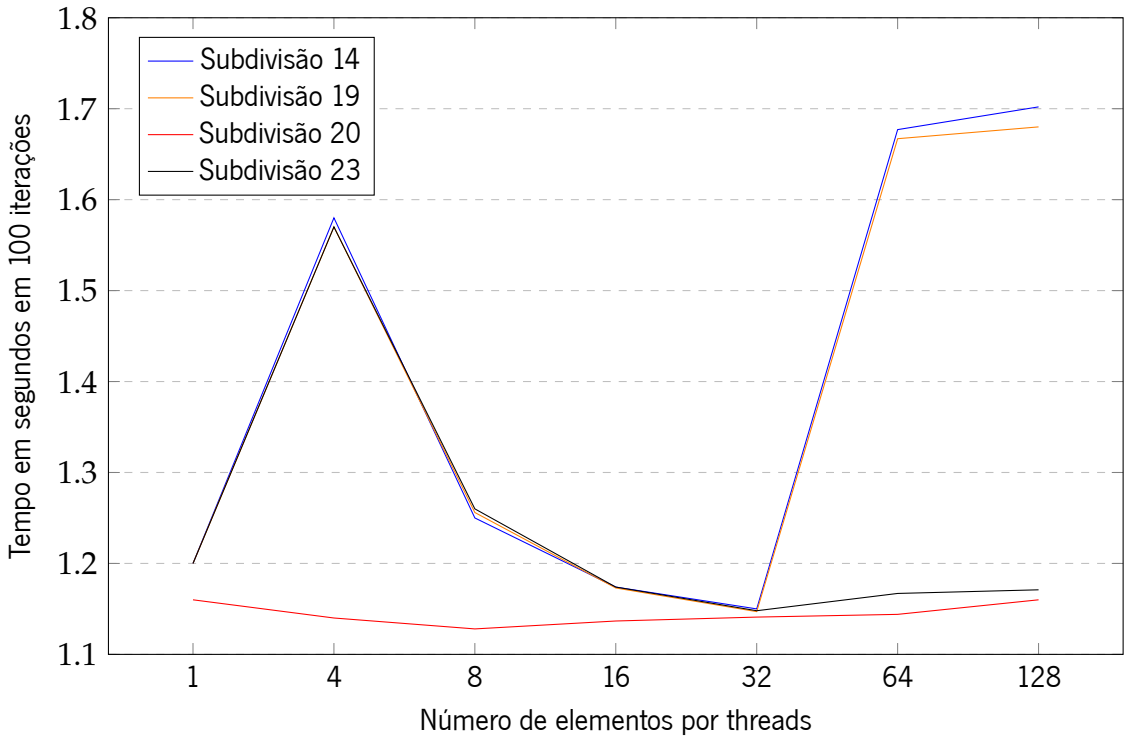


Figura 54: Gráfico da evolução do tempo de execução em relação ao aumento de elementos por thread no GPU - Subdivisões 14,19,20,23

8.7 Tempos finais

Os melhores tempos por subdivisão estão representados na tabela 14. Sendo o tempo total em 100 iterações de 38.62 segundos, sendo que cada iteração demora 0.3862 segundos por iteração. Estes resultados são 93% mais eficientes do que os resultados obtidos inicialmente, isto podendo significar que enquanto este código processa aproximadamente 13 imagens o código inicial processa apenas uma.

Tabela 14: Tempo de cada subdivisão em 100 iterações - Finais

Subdivisão	Tempo em segundos em 100 iterações	Subdivisão	Tempo em segundos em 100 iterações
1	0,0000725	13	0
2	1,289	14	1,15
3	3,78	15	0,817
4	0	16	0,00000763
5	4,39	17	1,97
6	0,967	18	0,66
7	7,57	19	1,173
8	4.829	20	1,128
9	1,13	21	1,97
10	1,97	22	0,66
11	0,66	23	1,148
12	1,36		

9

CONCLUSÃO E TRABALHOS FUTUROS

No âmbito desta dissertação, foi estudada a constituição do código que será utilizado na missão espacial PROBA-3. Sendo este um código que necessitava de otimização devido ao seu elevado tempo de execução. Para o estudo do mesmo foram utilizadas diferentes ferramentas para obter um melhor entendimento sobre como seria a melhor forma para o otimizar.

Inicialmente foi analisado o estado inicial do código e o compilador que estava a ser utilizado nesta fase, sendo este compilador o GFortran. Esta análise foi realizada através da verificação dos tempos de execução de cada subdivisão e de quais formas este compilador estava a influenciar o código com os métodos de otimização realizados pelo mesmo. Foi possível entender que apenas com a utilização de *flags* de otimização de um compilador é possível obter melhores resultados no código realizado.

Em seguida, foi utilizado outro compilador (PGI Fortran) para comparar as diferentes otimizações realizadas pelos dois compiladores e estudar quais destas seriam as mais benéficas para serem aplicadas no código inicial. Este compilador foi escolhido por ser um compilador da NVIDIA e suportar OpenACC e CUDA Fortran.

Após o estudo realizado ao comportamento do código ao ser compilado por dois compiladores diferentes, iniciou-se a análise à probabilidade de paralelizar o código. Inicialmente para obter resultados do código paralelizado foi utilizado OpenACC. O objetivo do uso de OpenACC foi o de estudar de forma menos complexa o diferente comportamento entre a execução das subdivisões nos dois diferentes processadores e o de estudar o impacto deste modelo na programação em paralelo. Com este, concluiu-se que a secção melhora consideravelmente com a utilização de mais do que uma *thread* do CPU, onde se obteve uma diminuição de 1.475 segundos por iteração com a utilização de 7 *threads* do CPU utilizado. Após realizar o estudo ao comportamento da paralelização com o CPU foi realizado o mesmo estudo para o GPU. Neste estudo percebeu-se que individualmente as subdivisões paralelizadas obtêm melhores resultados no GPU comparativamente com o CPU. Com a análise com OpenACC à paralelização das subdivisões para o GPU foi bastante claro o *bottleneck* no que diz respeito à programação para este processador, sendo este problema o elevado tempo necessário para realizar a comunicação e transferência de dados entre os dois processadores. Este problema foi atenuado com a utilização de diretivas do OpenACC que permitem ao programador

gerir a memória. Mesmo com esta gestão os resultados obtidos com a paralelização para o GPU foram menos eficazes do que os com a paralelização para o CPU.

Por fim, com o intuito de eliminar o máximo possível o tempo ocupado em transferências de memória durante cada iteração foi desenvolvido o código em CUDA. Nos testes realizados verificou-se o poder computacional que o GPU tráz para a computação de fórmulas matemáticas e que com uma boa gestão de memória entre os dois processadores este apresenta excelentes resultados. Considerando que a GPU utilizada fica aquém de muitas outras existentes na atualidade, o resultado obtido com a mesma comparativamente ao código inicial executado no CPU foi extraordinário. Sendo o resultado obtido com a GPU um tempo total por iteração de 0.3862 segundos e o original com o CPU de 5,74 segundos por iteração.

Como o objetivo desta dissertação era de o código desenvolvido estar presente na missão espacial PROBA-3, será necessário realizar testes na GPU a ser utilizada nesta missão. Estes testes servem para determinar quais as melhores configurações para a GPU. Tal como estudado, para obter melhores resultados em diferentes plataformas utilizando uma linguagem de programação como CUDA, existe a necessidade de programar o código especificamente para a máquina em que este será executado.

Sendo que esta dissertação se focou maioritariamente no desenvolvimento de código paralelo para ser executado no GPU e que apenas foi realizada uma curta análise da paralelização do código para CPU, seria também interessante realizar um estudo com a mesma profundidade para o CPU.

10

BIBLIOGRAFIA

[1] *Lyot, B. δ Marshall, R. K.„ The Study of the Solar Corona without an Eclipse, Journal of the Royal Astronomical Society of Canada, Vol. 27, p.265 1933*

[2] *Wlerick G. and Axtell J. , Solar activity cycle variation of the K corona,Astrophysical Journal, Part 1 (ISSN 0004-637X), vol. 285, p. 354-358., Oct. 1, 1984,*

[3] *Newkirk, Gordon, Jr.; Eddy, John A., A Coronograph Above the Atmosphere,1957*

[4] *Brueckner, G.E., Howard, R.A., Koomen, M.J. et al. The Large Angle Spectroscopic Coronagraph (LASCO). Sol Phys 162, 357–402 (1995).*

[5] *Koomen et al. Solar instruments on the P78-1 spacecraft Japan Society for the Promotion of Science and National Science Foundation, U.S.-Japan Seminar on the Recent Advances in the Understanding of Solar Flares, Tokyo, Japan, Oct. 5-8, 1982 Solar Physics (ISSN 0038-0938), 1975*

[6] *MacQueen et al.The Outer Solar Corona as Observed from Skylab: Preliminary Results. Astrophysical Journal, vol. 187, p.L85 ,1974*

[7] *MacQueen et al. Satellite Observations of the Outer Corona Near Sunspot Maximum. Proceedings of the International Astronomical Union 86:439-442, 1980*

[8] *Brueckner, G. E., Howard, R. A., Koomen, M. J., Korendyke, C. M., Michels, D. J.,Moses, J. D., The Large Angle Spectroscopic Coronagraph (LASCO), 1995*

[9] *S. V. Shestov and A. N. Zhukov, Influence of misalignments on the performance of externally occulted solar coronagraphs, 2018*

[10] *Wang, T., Reginald, N. L., Davila, J. M., St. Cyr, O. C., δ Thompson, W. T., Sol. Phys., 292, 97, 2017*

[11] *Shearer, P., Frazin, R. A., Hero, III., A. O., δ Gilbert, A. C. ApJ, 749, L8, 2012*

[12] *Goryaev, F., Slemzin, V., Vainshtein, L., δ Williams, D. R. 2014, ApJ, 781, 100, 2014*

[13] *Dolla, L., δ Solomon, J.,Solar off-limb line widths: Alfvén waves, ion-cyclotron waves, and preferential heating, AδA, 483, 271, 2008*

[14] *Högbom, J. A., Aperture Synthesis with a Non-Regular Distribution of Interferometer Baselines Astronomy and Astrophysics Supplement, Vol. 15, p.417 AδAS, 15, 417, 1974*

[15] *J.S.Llorente,A.Agenjoa,C.Carrascosa,C.de Negueruela, A.Mestreau-Garreau, A.Cropp, A.Santovincenzo, PROBA-3: Precise formation flying demonstration mission, 2017*

[16] *C. Aime , Theoretical performance of solar coronagraphs using sharp-edged or apodized circular external occulters , AδA 558, A138, 2013*

[17] *C.Aime ,Apodized apertures for solar coronagraphy , AδA 467, 317–325, 2007*

[18] *Dan Sirbu, Yunjong Kim, N. Jeremy Kasdin and Robert J. Vanderbei, Diffraction-based Sensitivity Analysis for an External Occulter Laboratory Demonstration, 2016*

[19] *Rougeot, R.; Flamary, R.; Galano, D.; Aime, C., Performance of the hybrid externally occulted Lyot solar coronagraph. Application to ASPIICS, AδA,599,A2, 2017*

[20] *https://www.zemax.com/products/opticstudio, pesquisado em 29/03/2020*

[21] *S. V. Shestov, A. N. Zhukov and D. B. Seaton, Modeling and removal of optical ghosts in the PROBA-3/ASPIICS externally occulted solar coronagraph, 05 February 2019*

[22] *https://developer.nvidia.com/cuda-zone, pesquisado em 13/05/2020*

[23] *https://www.khronos.org/opencl/, pesquisado em 3/01/2020*

[24] *Lauro Cássio Martins de Paula, CUDA VS. OPENCL: Uma Comparação Teórica E Tecnológica, 2014*

[25] *Kamran Karimi, Neil G. Dickson, Firas Hamze, A Performance Comparison of CUDA and OpenCL, 2010*

[26] *Jianbin Fang, Ana Lucia Varbanescu and Henk Sips ,A Comprehensive Performance Comparison of CUDA and OpenCL, 2011*

[27] *Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli ,High Performance Discrete Fourier Transforms on Graphics Processors, 2008*

[28] *https://en.wikipedia.org/wiki/Jacobi_method, pesquisado em 14/03/2020*

[29] *Chapman S. J., Introduction to Fortran90/95, McGraw-Hill, 1998*

[30] *https://gcc.gnu.org/fortran/, pesquisado em 15/01/2020*

[31] *http://www.mrao.cam.ac.uk/ pa/f90Notes/HTMLNotesnode44.html, pesquisado em 23/6/2020*

[32] *https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, pesquisado em 29/03/2020*

[33] *https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/, pesquisado em 4/02/2020*

[34] *https://www.pgroup.com/about/, pesquisado em 24/06/2020*

[35] *Michael Wolfe,OpenACC Kernels and Parallel Constructs,2012*

[36] Dan C. Marinescu, in Cloud Computing (Second Edition), 2018

[37] PGI Profiler User Guide , 2014

[38] http://www.nvidia.com/object/cuda_home.html, pesquisado em 30/06/2020