UNIVERSIDADE D
COIMBRA

João Manuel Moreira Rodrigues

CÁLCULO DE DOSE DE RADIAÇÃO EM CUDA

Dissertação de Estágio no âmbito do Mestrado em Engenharia Informática, especialização em Engenharia de Software orientada pelo Professor Evgheni Polisciuc e pelo Engenheiro Hugo Amaro e apresentada à Faculdade de Ciências e Tecnologia / Departamento de Engenharia Informática.

Julho de 2022

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

João Manuel Moreira Rodrigues

CÁLCULO DE DOSE DE RADIAÇÃO EM CUDA

Dissertação de Estágio no âmbito do Mestrado em Engenharia Informática, especialização em Engenharia de Software orientada pelo Professor Evgheni Polisciuc e pelo Engenheiro Hugo Amaro e apresentada à Faculdade de Ciências e Tecnologia / Departamento de Engenharia Informática.

Julho de 2022

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE Ð
COIMBRA

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

João Manuel Moreira Rodrigues

RADIATION DOSE CALCULATION IN CUDA

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Professor Evgheni Polisciuc and by Engineer Hugo Amaro and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July 2022

Agradecimentos

Gostaria de agradecer a todas as pessoas que contribuiram para a realização deste trabalho e elaboração desta dissertação.

Agradecer ao Professor Evgheni Polisciuc, o meu orientador do DEI, pela disponibilidade e apoio prestados durante o meu estágio, pelos notáveis conhecimentos partilhados, orientação e organização na realização da dissertação.

Agradecer ao Engenheiro Hugo Amaro, o meu orientador do IPN, que sempre se mostrou disponível para esclarecer dúvidas, dar opiniões e sugestões valiosas, sem esquecer o importante apoio ao longo de todo o trabalho realizado.

Gostaria também de agradecer ao IPNLis pela possibilidade de ter concorrido a uma bolsa de investigação ao longo do estágio – experiência bastante positiva e enriquecedora.

Não poderia deixar de referir a Universidade de Coimbra pela forma como fui acolhido.

E, por fim, agradecer à minha família e amigos, pela força, incentivo e apoio demonstrados ao longo de todo o meu percurso académico.

"Na vida não existem soluções. Existem forças em marcha:

é preciso criá-las e, então, a elas seguem-se as soluções."

Antoine de Saint-Exupéry

Resumo

Atualmente assiste-se a uma forte evolução das tecnologias, nomeadamente para fins medicinais, como por exemplo tratamento de doenças. A radioterapia é uma delas que tem apresentado uma grande evolução ao longo dos anos. Antes, o tratamento por radioterapia afetava um grande volume de órgãos e células normais em volta do alvo a incidir, um cancro ou tumor. Com os progressos efetuados, estes tratamentos têm uma maior precisão, o que diminui drasticamente o número e a intensidade dos efeitos colaterais. Hoje em dia, 60% dos doentes com cancro ou com tumores malignos realizam radioterapia.

Com este trabalho pretende-se implementar o algoritmo de cálculo de dose de radiação com recurso à aceleração de hardware GPU. Em particular, iremos implementar e estudar a multiplicação de matrizes de grandes dimensões com o objetivo de melhorar o tempo de cálculo.

Ao longo desta dissertação pretende-se explicar o planeamento de tratamentos por radioterapia, bem como analisar esses processos de planeamento, os equipamentos usados, as suas diferentes técnicas, as diferentes formas de cálculo de dose. O estudo destas questões é pertinente na medida em que, dependendo das decisões tomadas sobre cada detalhe e cada parâmetro durante o planeamento, conseguiremos obter uma melhor performance e, consequentemente, melhores resultados.

Palavras-Chave

Radioterapia; Aceleração GPU; Cálculo Matricial; Python; Sparse Matrix;

Abstract

Currently, there is a strong evolution of technologies, namely for medicinal purposes such as treatment of diseases. Radiotherapy is one of them that has shown great evolution over the years. Rather, treatment by preventive treatment will cause a large volume of normal organs and cells around the target to an incidence, a cancer or a tumor. With progress becoming more important, these treatments have, what drastically, the number and potency of side effects. Nowadays, 60% of patients with cancer or with malignant tumors, therapeutic therapy.

With this work we intend to implement or increase the radiation dose using GPU hardware installation. In particular, we will implement and study the multiplication of large matrices in order to improve calculus.

Throughout this dissertation, different treatment solutions used by radiation are explained, as well as these different equipments are studied, as well as their different techniques, such as ways of calculating dose. The study of these issues and the extent to which the results are relevant and each of the parameter determination decisions during the best performance, and we will achieve a better and consequently a better one.

Keywords

Radiotherapy; GPU acceleration; Matrix Calculation; Python; Sparse Matrix

Índice

Capítulo 1 Introdução **6**
1.1 Motivação 6
1.2 Enquadramento 6
1.3 Research Statement 7
1.4 Objetivos 7
1.5 Contributos esperados 7
1.6 Estrutura do documento 8
Capítulo 2 Conceito de Radioterapia **9**
2.1 O que é? 9
2.2 Processo 9
2.3 Equipamentos usados na Radioterapia 10
2.3.1 Acelerador Linear 10
2.3.2 Multileaf-Collimator (MLCs) 11
Capítulo 3 Revisão de Literatura **12**
3.1 Sistemas computacionais na radioterapia 12
3.1.1 Técnicas na radioterapia 13
3.1.2 Algoritmos aplicados na Radioterapia 16
3.2 Algoritmos para otimização baseado no GPU 18
3.2.1 Overview das API do GPU 18
3.2.3 Algoritmos 20
3.2.3 Multiplicação com Sparse Matrix 23
Capítulo 4 Trabalho Realizado **24**
4.1 Dados 24
4.2 Setup 24
4.3 Matlab 25
4.4 Implementações em CPU 26
4.4.1 NumPy 26
4.4.2 Ciclos "for" 27
4.4.3 Nested List 27
4.5 Implementações em GPU 28
4.5.1 Naive – CUDA Kernel 28
4.5.2 Naive – Fast Multiplication 29
4.5.3 Naive – CUDA Kernel sem If 30
4.5.4 Naive – Cálculo da Matriz por completo 31
4.5.5 Multiplicação de Sparse Matrix 32
4.6 Resultados 33
4.6.1 Implementações iniciais 33
4.6.2 Resultados Finais 39
4.7 Integração com sistema principal 41
Capítulo 5 Plano de Trabalho **44**
Capítulo 6 Conclusão **46**
Referências **47**

Acrónimos

Acrónimo	Significado
TPS	Treatment Planning System
SaaS	Software as a Service
GPU	Graphics Processing Unit
CPU	Central Processing Unit
KDE	Kernel Density Estimation
API	Application Programming Interface
TC	Tomografia Computorizada
MLC	Multileaf-Collimator
OPENGL	Open Graphics Library
CUDA	Compute Unified Device Architecture
TPB	Threads por Bloco

Lista de Figuras

Figura 1 - Acelerador Linear e os seus componentes 10

Figura 2 - Exemplo de um Collimator a estreitar vários feixes de partículas 11

Figura 3 - Folhas de placas colimadoras de metal pesadas de um Multileaf-collimator posicionadas para moldar a radiação de vários feixes 11

Figura 4 - Diferentes técnicas de radioterapia (por ordem da esquerda): 3DCRT, IMRT, VMAT. As cores representam diferentes intensidades onde os feixes de radiação incidem. 12

Figura 5 - Distribuição de Dose com 3DCRT através de diferentes ângulos e local de foco com uma cor de maior intensidade 13

Figura 6 - Exemplo de um tratamento com a técnica IMRT, através de vários feixes de radiação, onde o local de foco se apresenta com uma cor mais intensa 14

Figura 7 - Dose de radiação aplicada com VMAT, onde se nota que não há feixes de radiação num ângulo constante, minimizando a radiação dos órgãos em volta, como podemos verificar pela intensidade das cores 15

Figura 8 - Limitação do algoritmo de Pencil Beam em interfaces de diferentes densidades, como, por exemplo, um material ósseo 17

Figura 9 - Abordagem do Reduce Sum baseada em Árvore 20

Figura 10 - Exemplo de Kernel Density Estimaton com comparação de 3 diferentes funções $\varrho(x)$, $\psi(x)$, and $\psi(x)/x$: quadratic, Huber's, and Hampel's 21

Figura 11 - Desempenho de "speed-ups" de diferentes números de threads por bloco e tamanhos de ladrilhos sob os diferentes números de variáveis (do artigo de Michailidis et Margaritis, 2013) 22

Figura 12 - Representação de uma simples Matriz 23

Figura 13 - Representação de uma Sparse Matrix 23

Figura 14 - Excerto do código de cálculo de matrizes em Matlab 25

Figura 15 - Gráfico com tempos e média de execução do cálculo da multiplicação de matrizes em Matlab 26

Figura 16 – Excerto da implementação NumPy 26

Figura 17 - Implementação Ciclos ***For*** 27

Figura 18 - Implementação Nested List 27

Figura 19 - Excerto do código da implementação Naive - CUDA Kernel 28

Figura 20 - Excerto do código da implementação Naive - Fast Multiplication 29

Figura 21- Excerto do código da implementação Naive - CUDA Kernel sem *If* 30

Figura 22 - Naive – Cálculo da Matriz por completo 31

Figura 23 - Implementações da multiplicação Sparse Matrix no CPU e no GPU 32

Figura 24 - Gráfico de Tempo de Execução na Implementação NumPy 33

Figura 25 - Gráfico com uso de memória na Implementação NumPy 34
Figura 26 - Gráfico de Tempo de Execução na Implementação Ciclos ***For*** 34
Figura 27 - Gráfico de uso de memória na Implementação Ciclos ***For*** 35
Figura 28 - Gráfico de Tempo de Execução na Implementação Nested List 35
Figura 29 - Gráfico de uso de memória na Implementação Nested List 36
Figura 30 - Comparação da média do tempo de execução das 3 implementações 37
Figura 31 - Comparação da média do uso da memória das 3 implementações 37
Figura 32 - Gráfico de Tempo de Execução na Implementação Cálculo da matriz por completo 38
Figura 33 - Comparação entre os tempos de execução do cálculo no CPU e no GPU com Sparse Matrix 39
Figura 34 - Diferença entre os tempos de execução do cálculo no CPU e no GPU com Sparse Matrix 40
Figura 35 - Excerto do Código da integração com Python em Matlab 41
Figura 36 - Erro de o Matlab não suportar conversão de 'sparse double' para Python 41
Figura 37 - Nova solução de integração com Python 42
Figura 38 - Funções presentes na Package 43
Figura 39 - Plano de trabalho para o segundo semestre com as tarefas a realizar 44
Figura 40 - Plano de trabalho que foi realizado ao longo do segundo semestre 45

Capítulo 1
Introdução

Atualmente, a radioterapia é uma terapia muito popular com elevadas taxas de sucesso no tratamento de doentes com cancro ou tumores. O seu processo vai desde o planeamento ao cálculo da dose até à aplicação da radiação. No entanto, esta terapia enfrenta alguns desafios que limitam o número de tratamentos que cada instituição consegue aplicar num determinado espaço de tempo. Estes desafios são: o número de aceleradores de partículas disponíveis, a complexidade dos casos, a baixa eficiência do processo de criação de planos de tratamento e o número de dosimetrias, e equipamento para que estes possam realizar o seu trabalho, de que a instituição dispõe. O que pretendemos realizar com esta dissertação é uma otimização dos cálculos de dose necessários à criação de planos de tratamento de forma a tornar o processo de criação mais rápido e, consequentemente, aumentar performance; e, depois, testar essas implementações otimizadas no GPU e verificar se temos ou não alguma melhoria em comparação com as implementações até então conhecidas. Sendo este o objetivo de mais alto nível, o processo foi desde a implementação de multiplicação de matrizes até à comparação com as até então abordagens existentes e registar as suas performances.

1.1 Motivação

Nos dias de hoje, a tecnologia está presente na maioria das nossas ações. Na medicina, não deixa de ser igual, pois, em comunhão com a aplicação de métodos matemáticos, auxilia em muitas tarefas. E essa foi umas das razões que nos motivou para nos termos candidatado a esta dissertação. O desafio de otimizar o cálculo de dose de radiação para Treatment Planning System (TPS), com a aprendizagem de uma nova forma de implementação, em GPU, com estes todos elementos envolvidos, fez com que despertasse uma curiosidade sobre o assunto e, desta forma, queres conhecer mais a fundo e ter realizado este trabalho.

1.2 Enquadramento

Esta dissertação é integrada num projeto de investigação em copromoção, cujo objetivo consiste na implementação de um sistema, num modelo Software as a Service (SaaS), que exponha um conjunto de ferramentas e serviços para Médicos Radiologistas e Dosimetristas, desenhados para, por um lado, aproximar a oferta e procura de profissionais especializados e, por outro lado, agilizar o processo de construção de planos de tratamento para um determinado paciente.
Enquadra-se na construção de planos de tratamento, um dos mais importantes processos no tratamento por radioterapia. Este processo consiste em configurar um conjunto de parâmetros que irão determinar como irá ser aplicado o tratamento radioterapêutico ao doente. Esses parâmetros determinam o alinhamento do doente perante o equipamento emissor de radiação, o conjunto de feixes e respetiva direção e intensidade, que irão ser emitidos em direção ao volume alvo delineado pelo radioterapeuta. A construção dos planos de tratamento é, hoje em dia, feita com recurso a software especializado (Treatment Planning System ou TPS), que permite obter diferentes planos através da definição de um conjunto de parâmetros como pesos e

limites associados às diferentes estruturas de interesse (volumes a tratar e órgãos a poupar), sendo que alguns aspetos da configuração do tratamento podem ter de ser definidos à partida (como os ângulos ou arcos de irradiação). O tempo de planeamento de um tratamento depende da complexidade do caso a tratar.

1.3 Research Statement

Com a realização deste trabalho colocamos sob a hipótese o ganho substancial no cálculo de dose de radiação, delegando ao GPU a tarefa mais pesada do processo – o cálculo matricial. Em analogia aos casos de otimização através do recurso ao GPU, como por exemplo o Kernel Density Estimation (KDE), nós acreditamos que conseguimos ganho na performance devido à eficiência de GPU lidar com as tarefas de cálculo paralelo. Pretendemos ter um ganho de tempo e de performance no cálculo de dose para tratamentos de radioterapia. Para tal, ao longo desta dissertação, iremos testar diferentes algoritmos para cálculo com matrizes. Após realizar testes em CPU, a próxima fase é implementar em GPU e aí verificar se realmente conseguimos uma otimização na performance do cálculo ou não.

1.4 Objetivos

No início desta dissertação, foram definidos objetivos para serem realizados durante a mesma, em que o objetivo principal é o desenvolvimento de uma solução que consiga ter ganho, tanto de performance como de tempo, no cálculo de multiplicação de matrizes, em relação à solução atual. Para tal, foram realizados diversos passos para conseguir alcançar o objetivo final. Esses passos foram:

- Pesquisar sobre o tema Radioterapia. Desse modo, vamos estar mais dentro do tema e ter um maior conhecimento sobre ele, ganhando algumas bases;
- Realizar a implementação de leitura de dados de grande dimensão;
- Implementar e testar vários algoritmos de multiplicação de matrizes no CPU;
- Implementar e testar vários algoritmos de multiplicação de matrizes no GPU;
- Comparar a performance das várias implementações no CPU e no GPU;
- Elaborar de um Package com o algoritmo implementado em Python para poder distribuir o instalador do mesmo.

1.5 Contributos esperados

Com esta dissertação são esperados três diferentes contributos. Um deles é a implementação dos algoritmos. Estas implementações terão diferentes formas de calcular a multiplicação de duas matrizes. Após as implementações, a validação dos resultados e uma Package disponível para ser instalada.

1.6 Estrutura do documento

Este documento é constituído por 6 secções. Nesta primeira secção, fizemos uma breve introdução a este trabalho, onde apresentamos o contexto em que este se enquadra e os objetivos que pretendemos alcançar. Depois, na segunda secção, que é sobre Radioterapia, explica-se o seu conceito e como é o processo, desde o planeamento até ao tratamento, e também os equipamentos usados neste tipo de terapia. Na terceira secção, que é composta por duas subsecções, temos a Revisão da Literatura. A primeira secção é sobre as técnicas e algoritmos aplicados na radioterapia. Já a segunda tem uma breve descrição sobre Sparse Matrix e de API's do GPU e exemplos de algoritmos de otimização baseados no GPU. O Trabalho Final, que é descrito na quarta secção, é sobre os dados utilizados para o mesmo, o setup, os diferentes tipos de implementações e testes realizados na leitura de dados e na multiplicação de matrizes no CPU e no GPU, acompanhados por gráficos. De seguida, na quinta secção, temos o planeamento do trabalho definido para o segundo semestre e também o plano de trabalho do que foi realmente efetuado durante o segundo semestre. E, por fim, temos a conclusão, onde apresentamos as conclusões alcançadas com o decorrer dos trabalhos realizados ao longo do semestre e com os resultados objetivos através dos testes.

Capítulo 2
Conceito de Radioterapia

Neste capítulo iremos apresentar alguns conceitos relacionados com radioterapia e os processos associados. Também, na mesma secção, apresentamos as tecnologias e os equipamentos usados nos tratamentos com radiação.

2.1 O que é?

Segundo Jham et al. (2006) e Vidal et. al (2010), a radioterapia consiste na utilização da radiação ionizante como agente terapêutico para destruir células tumorais, sendo empregue uma dose de radiação pré-calculada num volume de tecido, por um período de tempo determinado, tentando evitar tecidos adjacentes, pois a regeneração tecidular far-se-á à custa deles. Este tipo de tratamento é associado a estágios mais avançados da doença, podendo a quimioterapia associar-se quando os doentes apresentam fatores de risco de recorrência. Contudo, a radioterapia pode ser aplicada a casos de modo isolado para pequenos tumores em estágio inicial. As radiações ionizantes podem ser corpusculares, por incidência de eletrões, protões ou neutrões, e também podem ser eletromagnéticas, através de fotões. (Da Silva et al., 2006).

Dependendo da localização da fonte de energia, a radioterapia pode ser de dois tipos: externa ou interna. A escolha entre os diferentes tipos de radioterapia é consoante as características do tumor, da localização e do objetivo do tratamento. Com os avanços tecnológicos obtidos nos últimos anos, a radioterapia tornou-se muito menos tóxica e mais efetiva (Mancini, 2020).

2.2 Processo

O processo de Radioterapia é definido pelo planeamento do tratamento, determinando a forma mais adequada de irradiar o paciente, especificando como e quais parâmetros serão utilizados. Este planeamento é a combinação das seguintes etapas (Wecare, 2018):

1. Escolher um posicionamento adequado do paciente e método de imobilização para que os tratamentos sejam reproduzíveis;
2. Identificar a forma e a localização do tumor (ou seja, o alvo) e dos órgãos vizinhos em risco;
3. Seleção de um feixe adequado;
4. Avaliação da distribuição da dose resultante;
5. Calcular as configurações da máquina de tratamento para fornecer a dose absoluta necessária.

A construção do plano de tratamento inicia-se, normalmente, com a realização de uma Tomografia Computorizada (TC). Geralmente, um tratamento por radioterapia consiste numa

série de sessões de tratamento, dividida por 4 a 5 vezes por semana, ao longo de várias semanas, com duração de 15 a 40 minutos. (Barra, 2020).

2.3 Equipamentos usados na Radioterapia

2.3.1 Acelerador Linear

O Acelerador Linear é, de longe, o equipamento mais comum nos tratamentos de radioterapia (Figura 1). É o mais versátil, permitindo fazer a vasta maioria dos tratamentos necessários. É um tipo de acelerador de partículas no qual partículas carregadas (eletrões, protões, iões) percorrem uma trajetória retilínea, diferentemente de aceleradores circulares (Nandi, 2004). A energia final das partículas é alcançada através de sucessivos incrementos na energia ao longo da aceleração. A radioterapia usa raios X que são produzidos quando os eletrões acelerados atingem um alvo de metal pesado. (Nandi, 2004). O tratamento de pacientes com acelerador linear tem fundamentalmente o seguinte processo:

1. O paciente é colocado na mesa de tratamento, onde acontecerá o tratamento que dura aproximadamente 15 minutos;
2. Com as luzes da sala de tratamento desligadas e auxílio do ODI (indicador ótico de distância) e lasers, localizam-se as marcas para tratamento no paciente, posicionando-o conforme no plano de tratamento;
3. É feita uma radiografia de verificação da área que receberá a terapia.
4. Com aprovação do médico, é iniciada a aplicação da radiação;
5. Terminado o tratamento, o paciente deixa a sala, devendo retornar no dia seguinte para a próxima aplicação (VARIAN, 1999).

Existem algumas versões dos aceleradores e a sua escolha dependerá sempre do caso clínico e do objetivo do tratamento. Estes tipos de equipamentos não existem em todos os centros de radioterapia. São equipamentos que representam investimentos avultados e cuja aquisição é integrada na realidade da instituição.

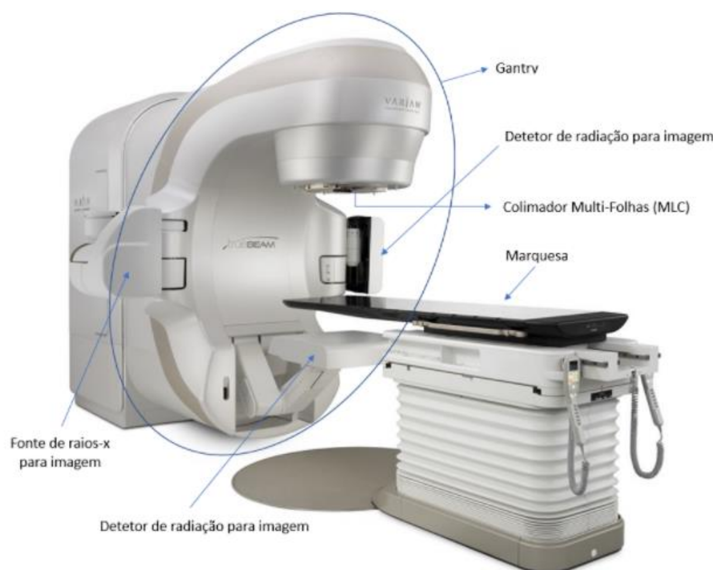

Figura 1 - Acelerador Linear e os seus componentes

2.3.2 Multileaf-Collimator (MLCs)

Um collimator é um dispositivo que estreita um feixe de partículas ou ondas (Figura 2). Estreitar pode significar tanto fazer com que as direções de movimento se tornem mais alinhadas em uma direção específica (isto é, fazer luz colimada ou raios paralelos), como fazer com que a secção transversal espacial do feixe fique menor (dispositivo de limitação de feixe), como é demonstrado na Figura 2 (Dexter, 2004). São usados nos aceleradores lineares para tratamentos de Radioterapia. Os collimators encontram-se posicionados imediatamente à frente dos emissores de partículas. Estes ajudam a moldar o feixe de radiação que emerge da máquina e podem limitar o tamanho máximo do campo de um feixe (Takahashi, 1965).

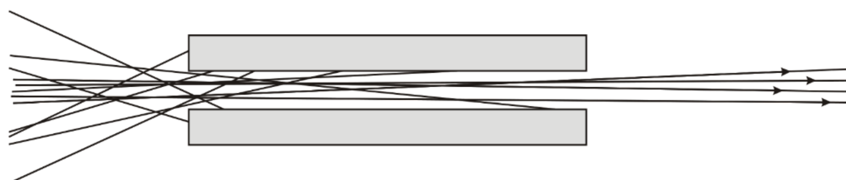

Figura 2 - Exemplo de um Collimator a estreitar vários feixes de partículas

Os Multileaf-Collimator (Figura 3) são um dispositivo limitador de feixe feito de "folhas" individuais de um material com numeração atómica alta, geralmente tungsténio, que se pode mover independentemente para dentro e para fora do caminho de um feixe de radiação para dar forma e variar sua intensidade (Galvin et all., 1993). São usados em radioterapia de feixe externo para fornecer moldagem conformada de feixes. Especificamente, a radioterapia conformada e a terapia de radiação modulada por intensidade (IMRT) podem ser administradas usando MLCs. Os Multileaf-Collimator são constituídos aproximadamente por 50-120 folhas de placas colimadoras de metal pesadas que deslizam no lugar para, como dito anteriormente, moldar os feixes de radiação (Galvin et all., 1993).

Figura 3 - Folhas de placas colimadoras de metal pesadas de um Multileaf-collimator posicionadas para moldar a radiação de vários feixes

Capítulo 3
Revisão de Literatura

Este capítulo foca-se na revisão das técnicas computacionais usadas na radioterapia. Começamos com as técnicas de aplicação de radiação. A seguir, falaremos sobre os algoritmos de cálculo de dose de radiação. Culminamos a revisão bibliográfica com dois exemplos de algoritmos que têm implementação no CPU e GPU.

3.1 Sistemas computacionais na radioterapia

Como referido anteriormente, a radioterapia externa é a forma mais comum, quando se fala de tratamentos com radiação, e é a utilizada neste projeto. Este tipo de tratamento consiste na aplicação de radiação, emitida por uma fonte externa ao corpo. Nesta secção iremos descrever e identificar as principais características das 3 diferentes tecnologias mais usadas nos tratamentos de radioterapia: 3DCRT, IMRT e VMAT (Figura 4).

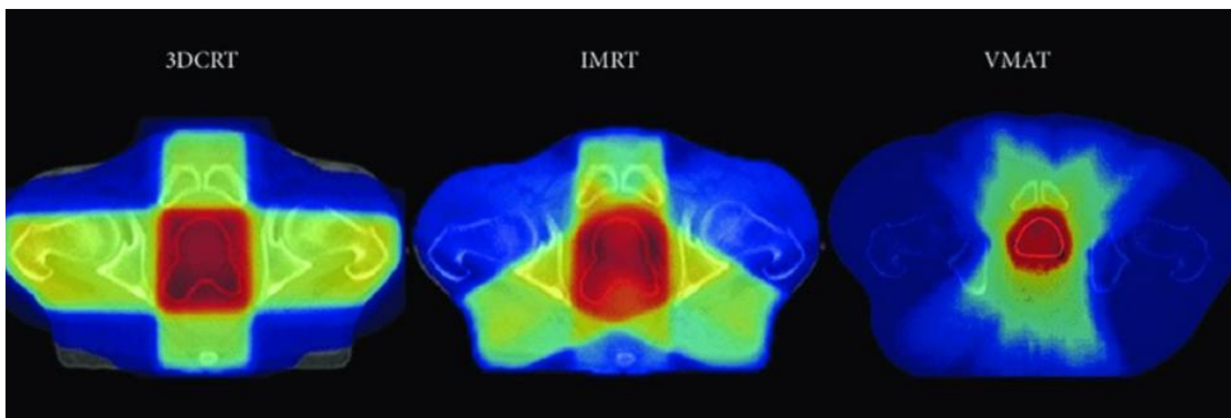

Figura 4 - Diferentes técnicas de radioterapia (por ordem da esquerda): 3DCRT, IMRT, VMAT. As cores representam diferentes intensidades onde os feixes de radiação incidem.

3.1.1 Técnicas na radioterapia

3.1.1.1 - 3DCRT

Segundo a Varian Medical Systems (2019), a técnica 3D Conformal Radiation Therapy é a mais utilizada. Tem como base uma imagem de uma Tomografia Computorizada (TC), onde são identificadas as áreas a tratar e as células/órgãos a proteger. Com estes dados, é efetuado o planeamento do tratamento. É uma técnica constituída por um ou mais feixes de radiação de incidência, através dos quais a dose é administrada. Esses feixes de radiação têm um ângulo particular face ao corpo, com proteções devidamente identificadas através do planeamento, e uma dose de radiação específica (Varian Medical Systems, 2019). Todos os feixes convergem na zona do tumor, vindos de direções diferentes, permitindo assim distribuir a dose de radiação. Deste modo, cada feixe de tratamento administra uma dose concreta de radiação que pode ser bastante diferente entre todos eles. Ou seja, a dose a administrar é dividida por todas as incidências (Varian Medical Systems, 2019). Cada feixe é "convertido" numa forma através de um Multi-Leaf Collimator presente no equipamento, no sentido de moldar o feixe à forma da área a tratar, e com o objetivo de proteger células e órgãos do paciente, sem prejuízo de a dose aplicar ao alvo (Varian Medical Systems, 2019).

Apesar de esta tecnologia demonstrar um impacto significativo, bem como na diminuição dos custos dos cuidados de saúde, a limitação reside no facto de ser influenciada pela probabilidade de complicação dos tecidos normais (Figura 5). Isto porque, embora haja distribuição do feixe de radiação para o tumor em várias direções e os campos angulados não terem de passar através de tanto tecido normal para atingir o volume alvo, verifica-se, em muitas ocasiões, que o aumento de conformação da dose ao volume tumoral e a capacidade de excluir os órgãos de risco não são ótimos, pelo que a 3DCRT pode não ter a capacidade de conseguir planos de tratamento satisfatórios (Monteiro, 2015).

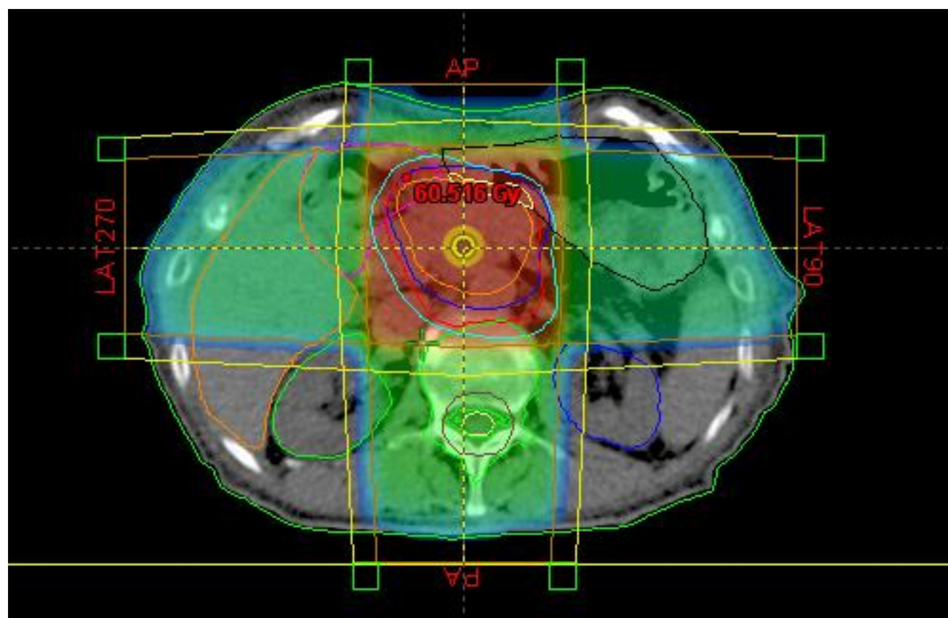

Figura 5 - Distribuição de Dose com 3DCRT através de diferentes ângulos e local de foco com uma cor de maior intensidade

3.1.1.2 - IMRT

Intensity-Modulated Radiation Therapy (IMRT) é uma evolução da técnica 3DCRT com algumas diferenças. Esta tecnologia foi considerada o mais excitante desenvolvimento em oncologia de radiação desde a introdução de imagens de tomografia computadorizada (TC) na criação do plano de tratamento (Bentzen, 2005; Nutting, 2003). Utiliza tecnologia avançada para manipular feixes de radiação de fotões e protões, para se conformar à forma de um tumor. Usa múltiplos pequenos feixes de fotões ou protões de intensidades variáveis para irradiar com precisão um tumor. A intensidade da radiação de cada feixe é controlada e a forma do feixe muda ao longo de cada tratamento.

A vantagem desta técnica é que se consegue adequar muito melhor o feixe de radiação à forma do volume do alvo e isso traduz-se em uma menor quantidade de órgãos e células normais vizinhas atingidas (Figura 6). Logo, menos efeitos secundários ou maior dose administrada com o mesmo nível de efeitos secundários.

Tal como na técnica anterior, o planeamento do tratamento tem como base uma imagem de TAC onde são identificados os volumes alvo e os órgãos adjacentes. No entanto, uma grande diferença surge nesta fase. A construção do planeamento é feita estabelecendo os objetivos finais, chamando-se, por isso, planeamento inverso. Neste tipo de planeamento, são as doses que determinam o feixe (Nicolucci, 2020) e é por tentativa, erro ou por experimentação, fazendo variar os objetivos para cada órgão e gerar um novo plano, a ver se é possível alcançar um mais vantajoso para o doente. É geralmente um planeamento mais complexo e mais demorado, sendo, por vezes, apelidada de técnica avançada. Na verdade, é uma técnica cada vez mais disponível e, por isso, cada vez mais usada, sendo já responsável pelo tratamento de uma boa parte dos doentes oncológicos em Portugal (Varian Medical Sistems, 2019).

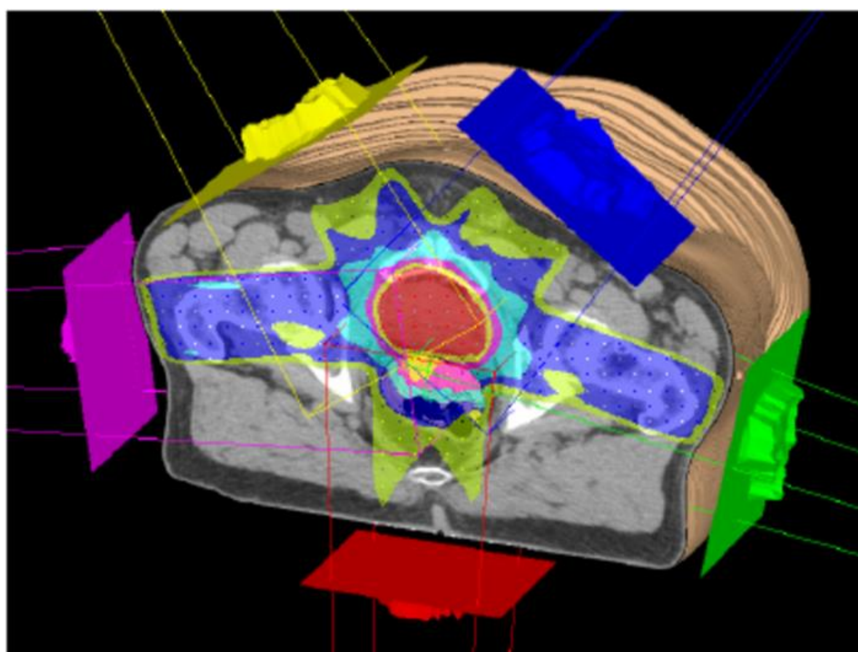

Figura 6 - Exemplo de um tratamento com a técnica IMRT, através de vários feixes de radiação, onde o local de foco se apresenta com uma cor mais intensa

3.1.1.3 - VMAT

É uma evolução da técnica IMRT. Juntando o potencial acrescido da rotação, esta técnica mostra-se bastante versátil, potencialmente diminuindo o tempo necessário para efetuar o tratamento. A sua vantagem é que, ao invés de termos algumas incidências, passamos a ter virtualmente 360, ou seja, fornece a dose de radiação continuamente, enquanto a máquina de tratamento gira (Figura 7). Para além disso, a velocidade a que o equipamento faz a rotação e a quantidade de radiação que sai em cada ponto também podem ser variáveis (Varian Medical Sistems, 2019).

Esta técnica molda com precisão a dose de radiação, minimizando a dose para os órgãos ao redor do tumor. É particularmente útil para tumores próximos a órgãos sensíveis e pode ser um tratamento eficaz para muitos tipos de cancro (Varian Medical Sistems, 2019).

A nível de capacidades, a técnica IMRT e a VMAT são bastantes semelhantes, sendo a maior vantagem da técnica VMAT a redução do tempo de administração do tratamento. Ao permitir um tratamento mais rápido, é diminuída a probabilidade de movimentos durante o mesmo, pois esta técnica exige um alto nível de precisão (Varian Medical Sistems, 2019).

Ao nível do planeamento, recorre-se também ao planeamento inverso. Tal como explicado na técnica anterior, este tipo de planeamento é feito estabelecendo um conjunto de objetivos para o tratamento (Varian Medical Sistems, 2019).

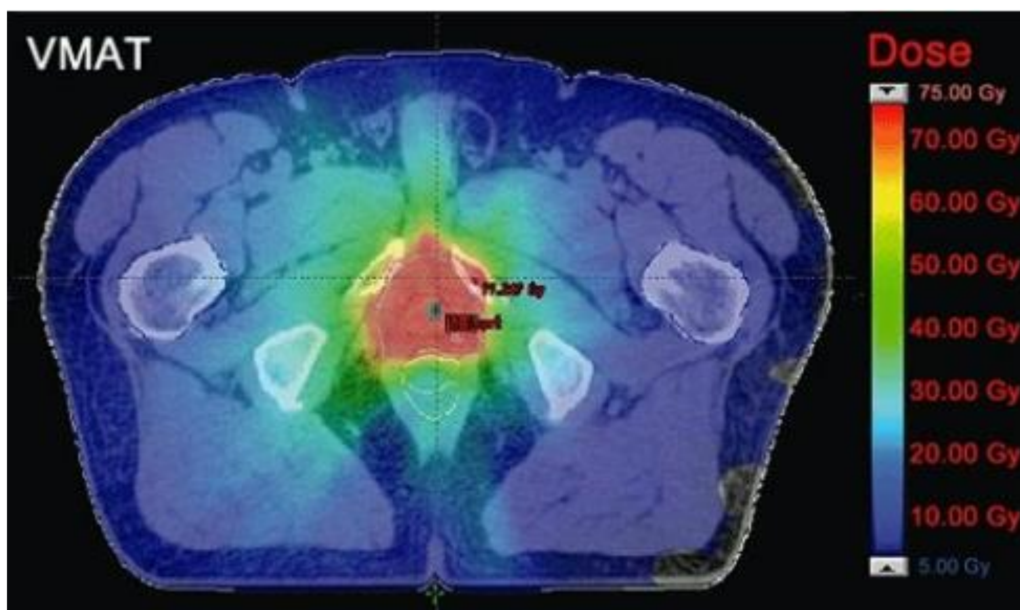

Figura 7 - Dose de radiação aplicada com VMAT, onde se nota que não há feixes de radiação num ângulo constante, minimizando a radiação dos órgãos em volta, como podemos verificar pela intensidade das cores

3.1.2 Algoritmos aplicados na Radioterapia

Nesta secção iremos abordar os algoritmos mais usados nos tratamentos de Radioterapia, pois estes são os que obtêm os melhores resultados. Para além disso, o algoritmo de Monte Carlo é o que consegue atingir resultados mais precisos para o cálculo da dose de radiação.

Monte Carlo

O algoritmo de Monte Carlo é um método de simulação estatística baseado em amostragem aleatória (J Med Signals Sens, 2011), que segue o caminho de cada partícula representativa através do acelerador, de modificadores de feixe e do paciente para determinar a dose, fluência e outras distribuições em pacientes. Usa física básica de interação de probabilidades (amostras por meio da seleção de números aleatórios) para determinar o destino das partículas representativas. Um número suficiente de partículas representativas são transportadas para produzir uma estatística de resultados aceitáveis (médias).

O método de Monte Carlo tem algumas preocupações, pois existem outros algoritmos suficientemente precisos também, e a experiência clínica atual resulta com base em alguns algoritmos imprecisos. Mas porque é que o Monte Carlo é uma boa opção para se aplicar na Radioterapia?

- Precisão atual de doses disponíveis de modelos de computação para planeamento de tratamentos de radiação são limitados;
- Discrepâncias em comparação com a dose real distribuições podem ser clinicamente significativas para muitos casos;
- As discrepâncias reveladas pelas previsões de dose podem ser corrigidas usando diferentes técnicas de tratamento, como, por exemplo, o uso de margens diferentes, energias de feixe e modulação de intensidade;
- A alta precisão agora é prática e acessível com simulações de Monte Carlo no transporte de radiação;
- Elimina tentativas e erros trabalhosos de parametrização e refinamento de modelos
- Reduz no tempo e na quantidade de dados de distribuição de dose medida necessários para a validação;
- Melhoria na qualidade da resposta da dose aos dados;
- Estimativa precisa de quantidades, difícil ou impossível de medir;

Um código do algoritmo de Monte Carlo deve considerar todos os aspetos do transporte de eletrões e fotões e deve ser capaz de produzir resultados precisos em um fantasma heterogéneo (J Med Signals Sens, 2011). Vários códigos do Monte Carlo de uso geral foram desenvolvidos para cálculo de transporte de radiação, que são usados na medicina, como EGS4 (Nelson W. et all., 1985), EGSnrc (Kawrakow et all., 2000), MCNP (Briesmeister J. F., 2000), entre outros. Os códigos de Monte Carlo para simulação de aceleradores lineares e cálculo de dose no paciente são BEAMnrc (Rogers D. W et all., 2004) e DOSXYZnrc (Walters B. R et all., 2005).

Pencil Beam

Pencil Beam é um algoritmo popular disponível em muitos sistemas comerciais de planeamento de tratamentos, que foi desenvolvido por Hogstrom e alguns colaboradores (Hogstrom et al., 1981; Hogstrom e Almond, 1983). Este método baseia-se na solução de Fermi-Eyges (Eyges, 1948) para a equação de transporte de eletrões, na qual é realizada uma distribuição de dose de feixe de lápis infinitamente estreita. Os grãos de dose são adquiridos num meio homogéneo, como água, para calcular a dose absorvida. O algoritmo representa um grande avanço em relação aos métodos empíricos anteriores, que são válidos apenas para inomogeneidades de 'laje' mais amplas do que o feixe incidente (Cygler et al., 1986). As distribuições de dose são previstas na presença de heterogeneidades menores pela soma das contribuições de feixes individuais de eletrões. A distribuição tecidual no paciente é definida através do uso de uma Tomografia Computadorizada.

A previsão precisa das distribuições de dose é especialmente importante nas regiões vizinhas da medula espinhal, que é propensa a lesões induzidas por radiação, uma séria complicação do tratamento para o paciente. Houve algumas avaliações sistemáticas do desempenho do algoritmo de Pencil Beam para pequenas heterogeneidades tridimensionais (Rogers et al., 1984; Nahum, 1985; Shortt et al., 1986).

O algoritmo tem algumas limitações para prever corretamente as doses quando são encontradas interfaces com materiais de densidades diferentes. Como é demonstrado na figura 8, dois subpontos do feixe de lápis de mesma energia têm alcances diferentes, dependendo se o eixo central passa do subponto ou não através do material ósseo (Bowen, 2018).

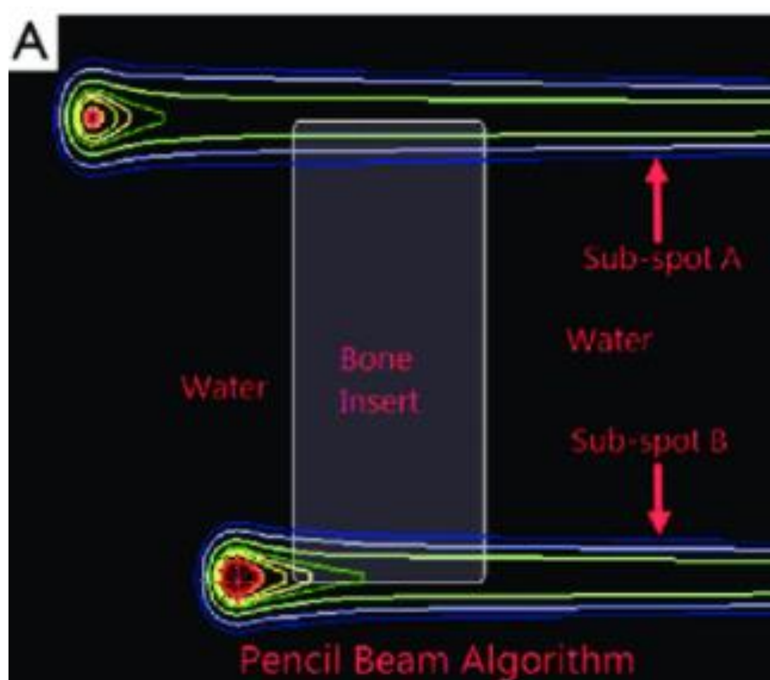

Figura 8 - Limitação do algoritmo de Pencil Beam em interfaces de diferentes densidades, como, por exemplo, um material ósseo

3.2 Algoritmos para otimização baseado no GPU

A possibilidade de programar unidades GPU para a computação de tarefas que não simplesmente a renderização gráfica, despertou o interesse de vários investigadores por volta do ano 2000 (Duarte, 2016). Numa primeira fase, quando os investigadores pioneiros dão os primeiros passos nesta área, a computação em GPU era realizada indiretamente: as operações tinham que ser mapeadas, primeiro, para um ambiente gráfico e, depois, manipuladas através de aplicações. Este panorama não é alterado até 2007, com o lançamento da linguagem de programação da NVIDIA dedicada exclusivamente à computação não gráfica das GPUs: CUDA (Duarte, 2016). Ao passar alguns algoritmos para GPU, temos a vantagem de utilizar a capacidade de computação paralela de uma GPU para alcançar uma alta eficiência, mantendo a mesma física de transporte de partículas e, portanto, o mesmo nível de precisão de simulação. A limitação é que na computação em GPU, a divergência de caminhos de execução entre threads pode reduzir consideravelmente a eficiência (X. Jia et all., 2011). Nas seguintes secções, vamos falar sobre dois exemplos de algoritmos baseados e implementados no GPU.

3.2.1 Overview das API do GPU

Nesta secção iremos fornecer uma revisão de três diferentes tecnologias atuais da computação gráfica, que são o OpenGL, CUDA e Vulkan. Esta revisão foi feita pois seria através de uma destas tecnologias que iriamos efetuar o cálculo de multiplicação de matrizes.

Open GL

O OpenGL (Open Graphics Library) é uma interface de software para hardware gráfico. A interface consiste em mais de 250 chamadas de funções diferentes que podem ser usadas para desenhar cenas complexas de duas e três dimensões a partir de primitivas geométricas simples, como pontos, linhas e polígonos. Há também rotinas para renderização das cenas com controlo sobre iluminação, propriedades da superfície do objeto, transparência, anti-aliasing e mapeamento de textura. O OpenGL foi projetado como uma interface simplificada e independente de hardware para ser implementada em muitas plataformas de hardware gráfico diferentes. É um software rápido, pois aproveita o hardware gráfico local e também portátil, pois está disponível em todas as principais plataformas de computação (Boston University TechWeb).

CUDA

O CUDA (Compute Unified Device Architecture) é uma API projetada para a computação paralela e para computação heterógena, criada pela NVIDIA. Esta plataforma dá a vários conjuntos de instruções virtuais de GPU e de computação paralela (Wikipédia, 2021). Apesar de ser uma plataforma que funciona com base em recursos da placa gráfica, é comum que se pense que CUDA sirva apenas para melhoria de jogos, contudo, pode ser um grande apoio ao CPU, ajudando a processar dados, como, por exemplo, a Harvard Engineering, a Harvard Medical School e o Brigham & Womens Hospital combinaram esforços para usar GPUs com o objetivo de simular o fluxo sanguíneo e identificar placas arteriais ocultas sem fazer uso de técnicas invasivas ou cirurgias exploratórias (TecMundo, 2011).

Um dos benefícios do uso de CUDA é a leitura paralela, em que o código pode ler endereços arbitrários na memória (Wikipédia, 2011). Mais uma vantagem é a memória compartilhada em que o CUDA expõe uma região de memória compartilhada rápida que pode ser compartilhada entre threads. Isso pode ser usado como um cache de usuário, permitindo maior largura de banda do que é possível utilizando textura lookups (Wikipédia, 2011). Outro benefício são os downloads mais rápidos e readbacks para a GPU e, por fim, tem suporte completo para operações de números inteiros e operações de bitwise (Abi-Chahla, 2008).

Apesar destas vantagens todas, o CUDA tem algumas limitações como, por exemplo, a renderização de texturas não é suportada, as cópias realizadas entre uma memória e outra podem gerar algum problema na performance das aplicações e só está disponível apenas para placas de vídeo fabricadas pela própria NVIDIA, diferente do Open GL (Wikipédia, 2011).

Vulkan

O Vulkan é uma API de baixo nível, que remove muitas das abstrações encontradas nas APIs gráficas da geração anterior. Isso é ótimo para fornecer desempenho máximo, mas coloca mais complexidade ao desenvolvedor. Contudo, há várias formas para ajudar a superar esses obstáculos e tornar produtivo rapidamente (vulkan.org). É uma API para acesso de multiplataforma de alta eficiência a GPU’s. O Vulkan inclui as mais recentes tecnologias gráficas, incluindo ray tracing, e está integrado aos drivers de produção da NVIDIA para soluções NVIDIA GeForce, RTX e Quadro no Windows e Linux, NVIDIA Shield e a plataforma de computação usando Android ou Linux (Nvidia Corporation, 2022).

3.2.3 Algoritmos

Reduce sum

Reduce Sum é um exemplo de uma implementação otimizada baseada no GPU. Este algoritmo de redução paralela combina uma matriz de elementos, produzindo, no fim, um único resultado. Um exemplo é o somatório de todos os elementos de um vetor ou matriz. É facilmente implementado em CUDA, pois o desempenho do CUDA tem as seguintes características: memória partilhada, onde CUDA expõe uma região de memória compartilhada rápida que pode ser partilhada entre threads e isso pode ser usado como um cache gerido pelo utilizador; e downloads e readbacks mais rápidos de e para a GPU. Há várias implementações com otimizações do Redusme Sum, onde pode atingir um tempo de execução 40x mais rápido, como, por exemplo, na utilização de múltiplos elementos por thread (Harris, 2007).

Este é um problema simples, com uma solução simples, que, como tal, funciona como um bom exemplo de otimização de uma solução a nível de performance (Harris, 2007). Esta abordagem é baseada em árvore, usada em cada bloco (Figura 9). Esta precisa de ser capaz de utilizar vários blocos de Thread para processar grandes arrays e para manter todos os multiprocessadores na GPU ocupados. Cada bloco de thread reduz uma parte do array (Harris, 2007). Ele funciona usando metade do número de threads dos elementos no conjunto de dados. Cada thread calcula o mínimo de seu próprio elemento e algum outro elemento. O elemento resultante é encaminhado para a próxima rodada. O número de threads é então reduzido pela metade e o processo repetido até que reste apenas um único elemento, que é o resultado da operação (Cook, 2018).

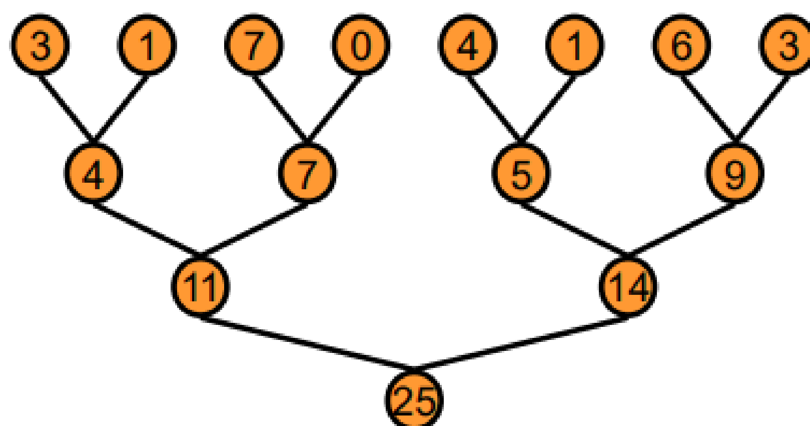

Figura 9 - Abordagem do Reduce Sum baseada em Árvore

Kernel Density Estimation

Kernel Density Estimation é um dos métodos não-paramétricos mais populares e tem sido usado com sucesso em um grande número de aplicações. Uma das aplicações famosas da Kernel Density Estimation é estimar as densidades marginais condicionais de classe de dados ao usar um classificador Bayes, o que pode melhorar a sua precisão. Estes tipos de métodos não requerem suposições distributivas e são dos mais importantes para análise e modelagem dos dados aplicados. Os métodos de Kernel Density Estimation são normalmente de complexidade computacional de O(n*2k), onde n é o número de observações e k o número de variáveis (Michailidis et Margaritis, 2013).

O principal problema dos métodos de Kernel Density Estimation são os enormes requisitos computacionais, especialmente para grandes conjuntos de dados. Uma maneira de acelerar esses métodos é usar o processamento paralelo.

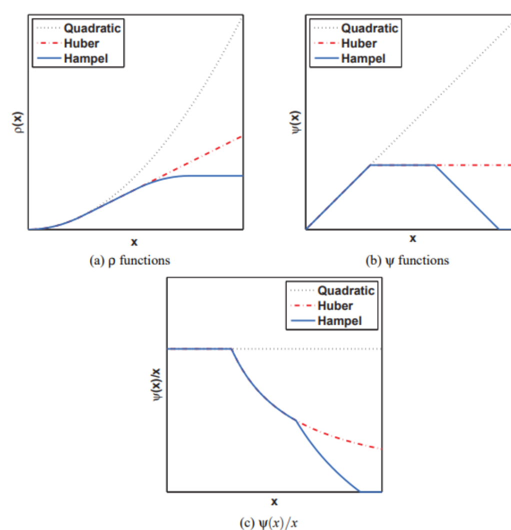

Figura 10 - Exemplo de Kernel Density Estimaton com comparação de 3 diferentes funções ϱ(x), ψ(x), and ψ(x)/x: quadratic, Huber's, and Hampel's

Um dos métodos do Kernel Density Estimation é o método Univariate. É um método eficaz para mostrar a estrutura de dados, não impõe uma forma paramétrica específica de densidade e também é fácil de começar (Ouyang, 2005). Em Univariate Kernel Density, a escolha da forma da função de Kernel não é particularmente importante, mas sim a escolha do valor da largura de banda (Ouyang, 2005).

O agendamento de trabalhos no GPU é dividido em vários níveis. E esses níveis são:

- Compute Divice (CD): um dispositivo de computação é, para todos os efeitos e propósitos, um indivíduo unidade de processamento; por exemplo, a CPU é um CD e também a GPU é um CD. Um trabalho do kernel pode ser distribuído por vários dispositivos.
- Compute Unit (CU): Um CD possui uma ou mais CU's e contém um ou mais elementos de processamento. Processando elementos em cada CU, compartilham parte da memória do hardware e unidades computacionais.
- Processing Elements (PE): Os elementos de processamento são os mais baixos diferenciáveis nível de processamento, efetivamente onde cada execução do kernel acontece (Amaro, 2015).

Um algoritmo otimizado em CUDA é baseado na memória partilhada. Através dos resultados obtidos por Michailidis et Margaritis (2013), verificou-se que o desempenho da otimização da memória partilhada para o método Univariate alcança ótimos resultados, particularmente para tamanhos de blocos que variam de 128 a 512, como mostra a figura 11.

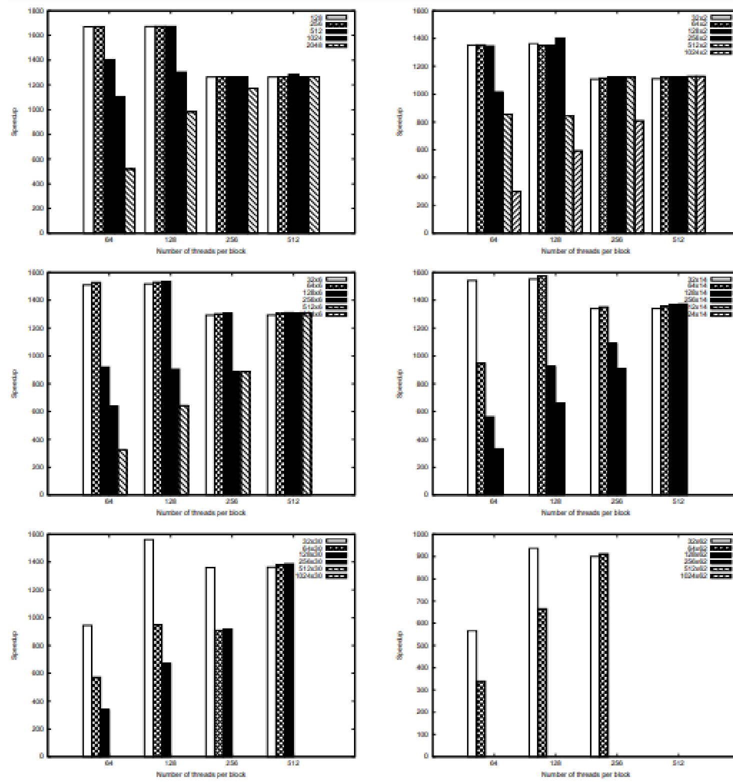

Figura 11 - Desempenho de "speed-ups" de diferentes números de threads por bloco e tamanhos de ladrilhos sob os diferentes números de variáveis (do artigo de Michailidis et Margaritis, 2013)

3.2.3 Multiplicação com Sparse Matrix

Sparse Matrix são matrizes m x n que tem uma reduzida quantidade de números não-zeros. O formato de uma linha Sparse compactada armazena apenas os índices da coluna e da linha e o referente valor diferente de zero (Yang, 2018). As suas vantagens em relação a uma simples matriz são o armazenamento e o tempo de computação. Como foi dito anteriormente, as Sparse Matrix contêm um número reduzido de números não-zeros comparada com uma matriz normal, logo vai ocupar menos memória para armazenar esses elementos. Em relação ao tempo de computação, só é preciso percorrer apenas os elementos diferentes de zero, o que reduz logicamente o seu tempo de computação.

As seguintes figuras (Figura 12 e Figura 13) vão ajudar a entender a representação de um Sparse Matrix em relação a uma simples Matrix.

	0	1	2	3
0	0	4	0	5
1	0	0	3	6
2	0	0	2	0
3	2	0	0	0
4	1	0	0	0

Figura 12 - Representação de uma simples Matriz

Table Structure

Row	Column	Value
0	1	4
0	3	5
1	2	3
1	3	6
2	2	2
3	0	2
4	0	1

Figura 13 - Representação de uma Sparse Matrix

Capítulo 4

Trabalho Realizado

Após ter sido feita a pesquisa sobre a radioterapia, as técnicas usadas e os diferentes algoritmos, chega a fase do trabalho realizado. O plano para este semestre era fazer diferentes implementações para o cálculo de duas matrizes, que está presente no cálculo de dose de radiação para tratamentos de radioterapia. No fim de as implementar, o objetivo era testar e ver qual deles tinha melhor performance, tanto no tempo como na ocupação da memória durante a execução. Por fim, intencionou-se implementar o código na sua versão final que pudesse ser instalada e usada dentro do ambiente do Python, através de uma Package.

4.1 Dados

Para a realização das implementações e dos testes, foi usado um dataset real, que nos foi fornecido por uma outra equipa, que também está neste projeto a trabalhar na geração automática de dosimetrias. Vieram em formato de um ficheiro *.mat*, que é o “HeadAndNek.mat”. Neste ficheiro encontramos vários tipos de matrizes. A maior delas é a matriz de Dose, que é a matriz utilizada no cálculo das implementações, é uma ***sparse double*** com 1736707x5108 de dimensão. Existem também vetores, um para cada estrutura interna relevante da zona do corpo do paciente, que vai ser incidido com radiação. Estes vetores contêm os índices de uma matriz que identificam os Voxels que compõem essas estruturas. No total, o tamanho da soma de todas as matrizes e vetores é cerca de 17,4GB.

O problema que vamos tentar resolver é otimizar a leitura dos dados para a memória e a multiplicação das matrizes, neste caso, da matriz de Dose com um vetor.

4.2 Setup

Para a realização das implementações e testes, foi utilizado um computador pessoal com as seguintes especificações:

Nome: Micro-Star International Co., Ltd. Katana GF66 11UC

Processador: 11th Gen Intel® Core™ i7-11800H @ 2.30GHz 8 cores

Gráfica: NVIDIA GeForce RTX 3050 Laptop GPU

Memória GPU: 1 GB

RAM: 16 GB

Sistema Operativo: Windows 10 Home 64 bits

Os principais testes foram efetuados com dois diferentes scripts, um que calcula o resultado da multiplicação de matrizes em GPU e outro que calcula em CPU, através da

Implementação que calcula com Sparse Matrix. As variáveis utilizadas para a realização dos testes eram a Matriz de Dose, que se mantinha sempre, e uma lista com vetores, em que, aumentando o número de vetores, de modo a poder verificar qual das implementações iria ser mais rápida, dependendo do número de vetores pela qual a Matriz de Dose ia multiplicar, o número de vetores ia sendo incrementado de 5 em 5 para termos uma melhor amostra de valores para comparação. Esses valores são diretamente guardados em Excel, para que, deste modo, seja mais fácil, no fim, podermos realizar diferentes tipos de gráficos.

4.3 Matlab

Devido a este projeto estar a ser desenvolvido em Matlab, foi implementado (Figura 14) e testado a multiplicação de matrizes em Matlab, de modo a ter de início valores para posteriormente serem comparados com outras implementações. Para isso, passamos o vetor que multiplica pela matriz de dose para uma *"Sparse Double Matrix".* O uso das Sparses Matrix terá uma melhor performance pois ao converter para este tipo de matriz, retira todos os elementos com valor zero, diminuído o número de cálculos para futuras multiplicações.

```
tic;
load('HeadAndNeck.mat');
vetor = sparse(double(vetor_x));
result = dose_matrix * vetor;
toc;
writematrix(result,'result_Matlab10.csv','Delimiter','comma');
```

Figura 14 - Excerto do código de cálculo de matrizes em Matlab

Após isso, foi corrida 10 vezes a implementação, guardando o tempo de execução de cada uma. Verificou-se que os tempos de execução variavam entre 2 segundos e 2,10 segundos, registando um tempo médio de 2,0566, com um desvio padrão de 0,0184. Todos os valores estão presentes na Figura 15.

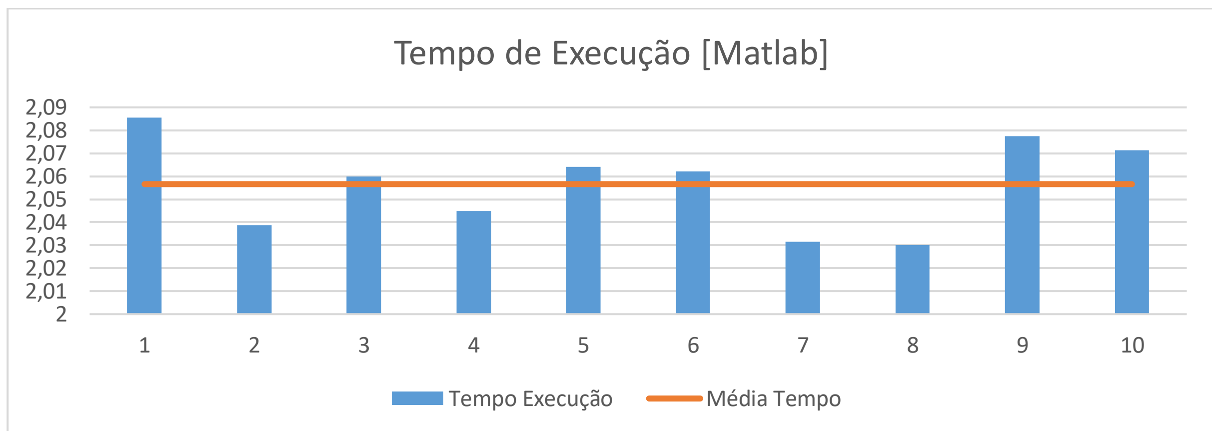

Figura 15 - Gráfico com tempos e média de execução do cálculo da multiplicação de matrizes em Matlab

4.4 Implementações em CPU

As 3 implementações que, de seguida, serão apresentadas, apenas diferem na forma como é feita a multiplicação de matrizes. Numa delas é utilizada a biblioteca Numpy, noutra recorremos aos ciclos ***for*** e, na última, foram utilizadas Nested Lists.

Primeiro, fazemos a leitura de todos os dados a partir do ficheiro "HeadAndNeck.mat". A partir dos dados recolhidos, guardamos a Dose Matriz, que é necessária para o cálculo da multiplicação de matrizes, numa Sparse Matrix. A seguirm faz-se o mesmo para o vetor x, com a pequena diferença de percorrer um ciclo ***for***, de modo a guardar os valores num array. Após termos as duas matrizes, convertemo-las para numpy arrays, para serem utilizadas nos diferentes cálculos de multiplicação de matrizes.

4.4.1 NumPy

Nesta primeira implementação, é utilizada a biblioteca NumPy (Figura 16). Foi criada em 2005, por Travis Oliphant e é uma biblioteca direcionada para trabalhar com array. A razão para termos implementado com o NumPy é porque é 50x mais rápido que as listas tradicionais de python porque as matrizes NumPy são armazenadas em um local contínuo de memória, para que assim possam ser acedidas e alteradas com mais eficiência. Este tipo de comportamento é chamado de localidade de referência em Ciência de Computação. O array em Numpy é o ***ndarray*** e fornece muitas funções de suporte que facilitam o trabalho.

Neste caso, para o cálculo de multiplicação de duas matrizes, usamos a função numpy.dot(), que retorna automaticamente uma matriz do produto de duas.

```
#Matriz 1736707x1
resultado = np.dot(matriz_A, np_vetor1)
```

Figura 16 – Excerto da implementação NumPy

4.4.2 Ciclos “for”

Já nesta implementação (Figura 17), usamos ciclos ***for*** para calcular a multiplicação de cada linha com cada coluna e guardar o resultado na posição correta no resultado final. É esperado que este tipo de implementação seja mais lento que o anterior, visto que tem uma complexidade temporal de $O(n^3)$, pois é uma multiplicação ingénua de duas matrizes n×n, calculando correlação parcial.

```
resultado = [[0]*1]*1736707

for i in range(len(matriz_A)):
    for j in range(len(vetor_X[0])):
        for k in range(len(vetor_X)):
            resultado[i][j] += matriz_A[i][k] * vetor_X[k][j]
```

Figura 17 - Implementação Ciclos ***For***

4.4.3 Nested List

Nesta última implementação (Figura 18), calculamos a multiplicação de matrizes através de Nested Lists, usando a feature Zip em python. Nested Lists são listas de listas ou de qualquer lista que tenha outra lista como elemento. Para esta implementação é útil, pois cria uma matriz e armazena os dados.

```
resultado = [[sum(a * b for a, b in zip(A_row, B_col)) for B_col in zip(*vetor_X)] for A_row in matriz_A]
```

Figura 18 - Implementação Nested List

4.5 Implementações em GPU

Após realizar as implementações, tanto em Matlab como em Python no CPU, foram feitas as implementas em GPU, utilizando CUDA. Tal como no CPU, foram realizadas 3 diferentes implementações. Tal como no CPU, começamos por ler os ficheiros "HeadAndNeck.mat" para obter as matrizes e os vetores necessários. Após isso, passa-se esses elementos para a gráfica, através da biblioteca CuPy, com a instrução cupy.array(). Esta é uma biblioteca destinada para computação acelerada em GPU com python, utilizando CUDA, e assim acelera algumas das operações em mais de 100x. Depois, é definido o valor de Threads por bloco (TPB) e do valor dos blocos por grelha. Por fim, chama-se a função principal, enviando os 3 arrays necessários para os cálculos.

4.5.1 Naive – CUDA Kernel

Nesta primeira implementação, é usado CUDA Kernel para a multiplicação de matrizes (Figura 19). Esta é uma implementação direta e intuitiva, contudo, é esperado que não tenha um desempenho muito bom, visto que os elementos da matriz são carregados várias vezes na memória, o que torna a execução mais lenta.

```
@cuda.jit
def matmul(A, B, C):
    """Perform square matrix multiplication of C = A * B
    """
    i, j = cuda.grid(2)
    if i < C.shape[0] and j < C.shape[1]:
        tmp = 0.
        for k in range(A.shape[1]):
            tmp += A[i, k] * B[k, j]
        C[i, j] = tmp
```

Figura 19 - Excerto do código da implementação Naive - CUDA Kernel

4.5.2 Naive – Fast Multiplication

Nesta implementação (Figura 20), espera-se que seja mais rápida que a anterior, pois usa memória partilhada para threads em um bloco para computar cooperativamente uma tarefa, como se pode reparar no ciclo for que está presente no código da figura X. Usa o método cuda.syncthreads(), de modo a esperar que todas a threads acabem de carregar.

```python
@cuda.jit
def fast_matmul(A, B, C):
    # Define an array in the shared memory
    # The size and type of the arrays must be known at compile time
    sA = cuda.shared.array(shape=(TPB, TPB), dtype=float32)
    sB = cuda.shared.array(shape=(TPB, TPB), dtype=float32)

    x, y = cuda.grid(2)

    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    bpg = cuda.gridDim.x    # blocks per grid

    if x >= C.shape[0] and y >= C.shape[1]:
        # Quit if (x, y) is outside of valid C boundary
        return

    # Each thread computes one element in the result matrix.
    # The dot product is chunked into dot products of TPB-long vectors.
    tmp = 0.
    for i in range(bpg):
        # Preload data into shared memory
        sA[tx, ty] = A[x, ty + i * TPB]
        sB[tx, ty] = B[tx + i * TPB, y]

        # Wait until all threads finish preloading
        cuda.syncthreads()

        # Computes partial product on the shared memory
        for j in range(TPB):
            tmp += sA[tx, j] * sB[j, ty]

        # Wait until all threads finish computing
        cuda.syncthreads()

    C[x, y] = tmp
```

Figura 20 - Excerto do código da implementação Naive - Fast Multiplication

4.5.3 Naive – CUDA Kernel sem If

Por último, esta é a implementação (Figura 21) mais simples e apenas difere numa linha de código em relação à primeira, que é no If. Como tal, é esperado que os resultados comparativamente com essa implementação não difiram muito.

```
@cuda.jit
def matmul(A, B, C):

    i, j = cuda.grid(2)
    tmp = 0.
    for k in range(A.shape[1]):
        tmp += A[i, k] * B[k, j]
    C[i, j] = tmp
```

Figura 21- Excerto do código da implementação Naive - CUDA Kernel sem *If*

4.5.4 Naive – Cálculo da Matriz por completo

Após alguns testes com as três anteriores implementações, reparámos que, com a matriz de dose que nos foi fornecida, não era possível correr a implementação, pois não havia memória suficiente. A solução encontrada foi dividir a matriz em várias partes de modo a conseguir multiplicá-la por completo, como mostra a seguinte figura (Figura 22).

```
nTimes = math.ceil(matrizDose.shape[0]/80000)

# Now start the kernel
inicio = time.time_ns()
for i in range(nTimes):
    if(i == 0):
        matmul[blockspergrid, threadsperblock](matrizDose_GPU, vetorX_GPU, w)
        wu0 = w.copy_to_host()
        w0 = cp.array(wu0)

        # Libertar memória
        matrizDose_GPU = None
        w = None

    elif(i == nTimes-1):
        w = cuda.device_array((80000, vetorX_GPU.shape[1]))
        matrizDose_GPU = cp.array(matrizDose[i*80000: , : ].toarray())
        matmul[blockspergrid, threadsperblock](matrizDose_GPU, vetorX_GPU, w)
        globals()['wu%s' % i] = cp.append(globals()['w%s' % (i-1)], w.copy_to_host(), axis = 0)

        # Libertar memória
        matrizDose_GPU = None
        w = None
        globals()['w%s' % (i-1)] = None
        globals()['wu%s' % (i-1)] = None

    else:
        w = cuda.device_array((80000, vetorX_GPU.shape[1]))
        matrizDose_GPU = cp.array(matrizDose[i*80000:(i+1)*80000, : ].toarray())
        matmul[blockspergrid, threadsperblock](matrizDose_GPU, vetorX_GPU, w)
        globals()['wu%s' % i] = cp.append(globals()['w%s' % (i-1)], w.copy_to_host(), axis = 0)
        globals()['w%s' % i] = cp.array(globals()['wu%s' % i])

        # Libertar memória
```

Figura 22 - Naive – Cálculo da Matriz por completo

Ou seja, inicialmente é calculado o número de vezes que será necessário correr o ciclo *for* para conseguir calcular a matriz totalmente, baseando no número máximo de linhas que podem ser multiplicadas no tamanho específico de memória. Depois, ao entrar no ciclo *for*, vai-se adicionar a multiplicação efetuada ao que até já foi feito e, assim sucessivamente, libertando a memória de variáveis que passam a não ter uso. Para realizar a multiplicação, é usada uma das funções anteriormente apresentadas.

4.5.5 Multiplicação de Sparse Matrix

Esta implementação foi baseada na implementação no Matlab, em que, para efetuar o cálculo, é necessário passar o vetor para Sparse Matrix para poder multiplicar com a matriz de dose.

Assim, esta implementação ficou semelhante à implementação do Numpy. A diferença é que, ao invés de estarmos a multiplicar dois numpy arrays, estamos a multiplicar duas sparse matrix. Esta implementação, tanto dá para calcular no CPU e no GPU. Para ser no GPU, basta enviar os dados para a memória gráfica, através do CuPy.

```
def multiply_Matrix_CPU(mat_sparse_dose, arrayVetores, NV):

    resultados = []

    #for matrix sparse calculation on the cpu
    for l in range(len(arrayVetores)):
        resultado = mat_sparse_dose.dot(csr_matrix(arrayVetores[l]))
        resultados.append(resultado)

    return resultados
```

```
def multiply_Matrix_GPU(mat_sparse_dose, arrayVetores, NV):

    mat_dose_gpu = cpx.scipy.sparse.csc_matrix(mat_sparse_dose)
    resultados = []

    #for matrix sparse calculation on the gpu
    for i in range(len(arrayVetores)):
        vector_host = np.float64(arrayVetores[i])
        vector_gpu = cp.asarray(vector_host)
        resultado = mat_dose_gpu.dot(vector_gpu)
        resultados.append(resultado)

    return resultados
```

Figura 23 - Implementações da multiplicação Sparse Matrix no CPU e no GPU

4.6 Resultados

Para a realização dos testes, foi executada 10 vezes cada implementação. Com isto, obtivemos o tempo de cada execução e, de cada implementação, a sua média respetiva e a memória usada em cada uma das diferentes implementações. Os resultados obtidos estão descritos nas seguintes secções.

4.6.1 Implementações iniciais

Numpy

Nesta primeira implementação, os resultados obtidos são significativamente bons porque, como foi dito na secção de implementação com NumPy, já era esperado que fosse a implementação mais rápida visto que a utilização da biblioteca NumPy iria ser 50x mais rápida do que com listas tradicionais. Quanto aos tempos de execução, a maioria deles foram abaixo de 60 segundos, no que resultou numa média aproximada de 53,775 segundos.

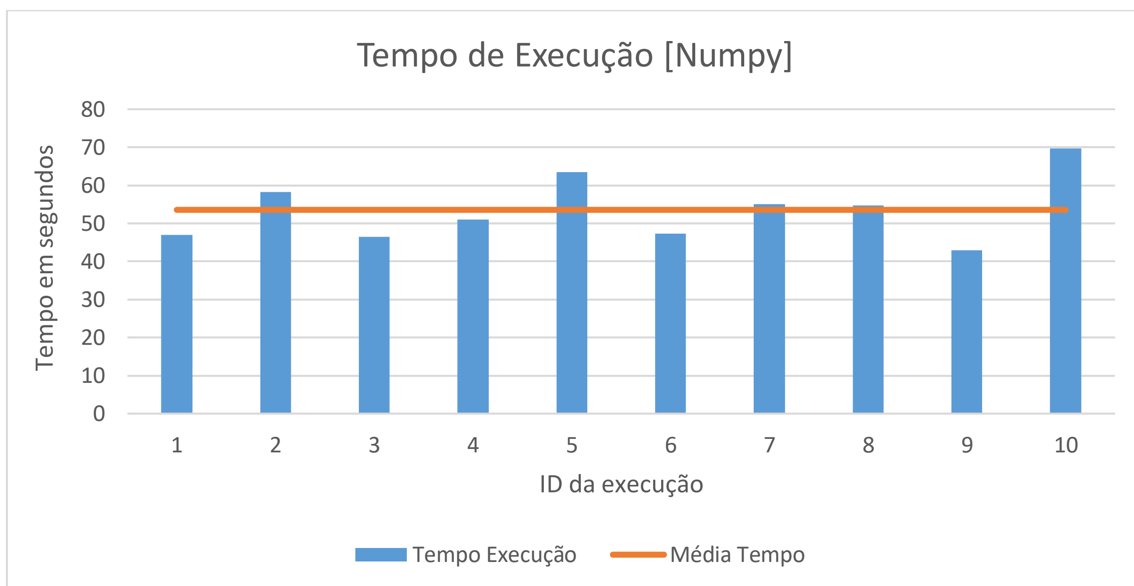

Figura 24 - Gráfico de Tempo de Execução na Implementação NumPy

Quanto à memória utilizada, a execução da implementação usou em média 12,893GB, utilizando cerca de 82% da memória do computador onde foram realizados os testes.

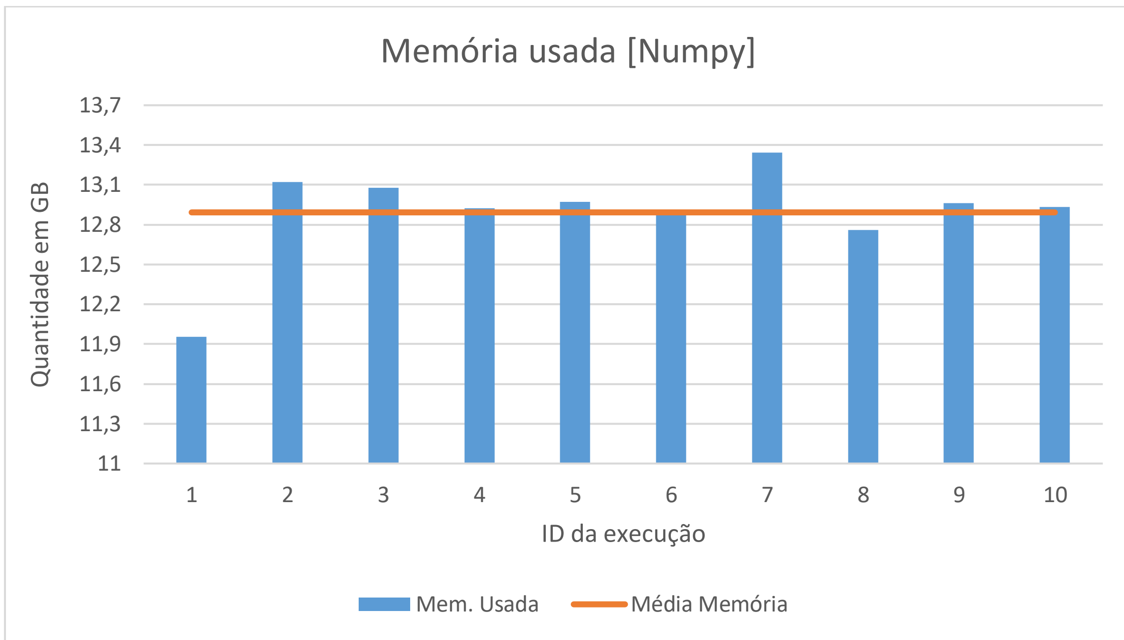

Figura 25 - Gráfico com uso de memória na Implementação NumPy

Ciclos "for"

Nesta implementação, foi onde as execuções demoraram mais tempo, ficando com uma média de 3579,83 segundos, correspondendo a aproximadamente 1 hora. Estes resultados confirmam o que já era esperado, como sendo a implementação mais lenta.

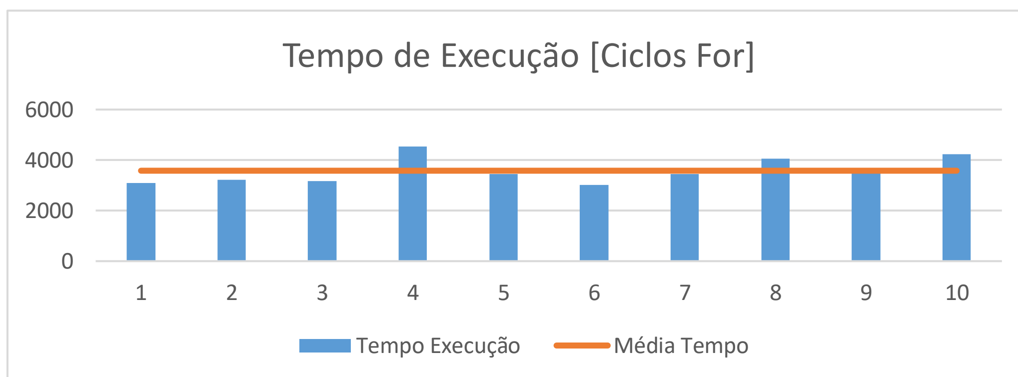

Figura 26 - Gráfico de Tempo de Execução na Implementação Ciclos ***For***

Quanto ao uso de memória, teve várias oscilações, desde aproximadamente 7GB até 11GB, sendo a médias de 8,5GB de utilização.

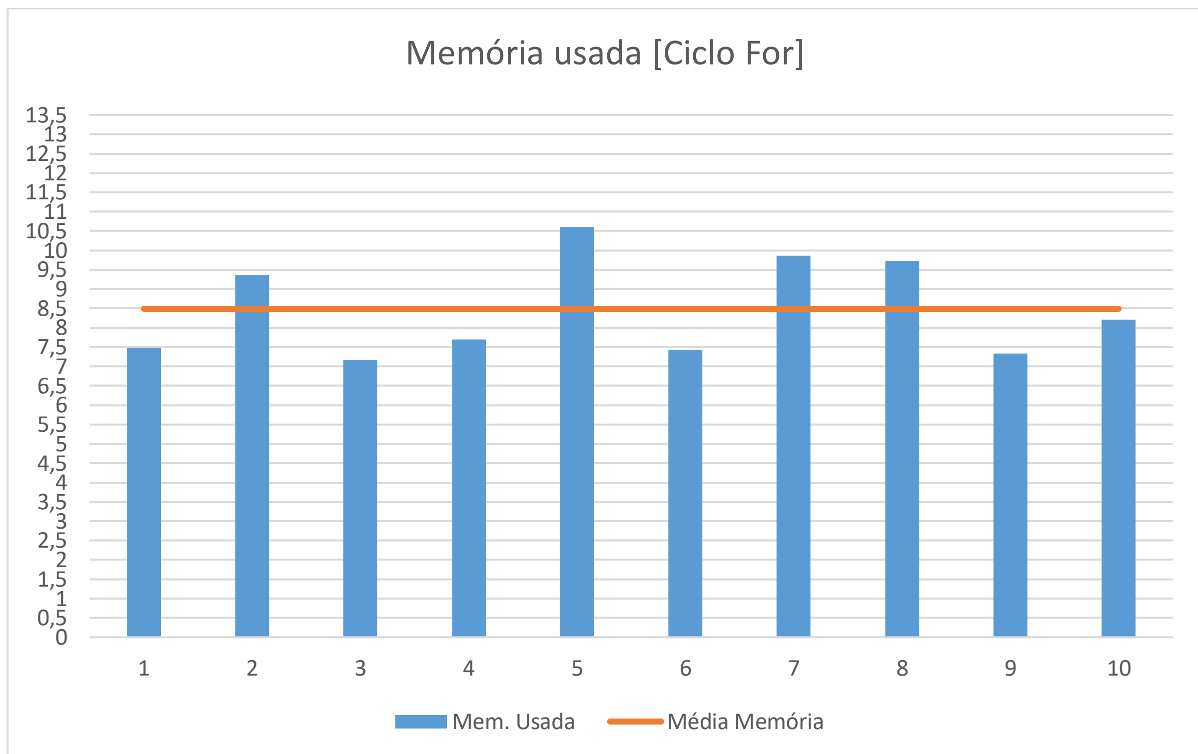

Figura 27 - Gráfico de uso de memória na Implementação Ciclos ***For***

NestedList

Nesta implementação , tal como na anterior, o tempo de execução também foi elevado, em média 46 minutos. Só cerca de 40% das execuções, foram abaixo da média de todas as execuções.

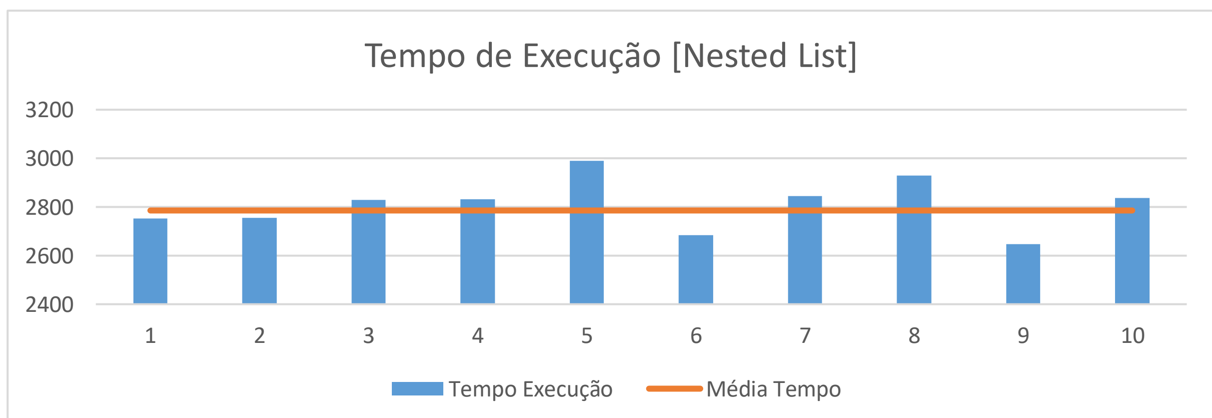

Figura 28 - Gráfico de Tempo de Execução na Implementação Nested List

Quanto ao uso da memória, verifica-se muito pouca oscilação, com quase todas as execuções muito perto da média. Só apenas a execução 1 e 3 é que tem uma maior diferença da média.

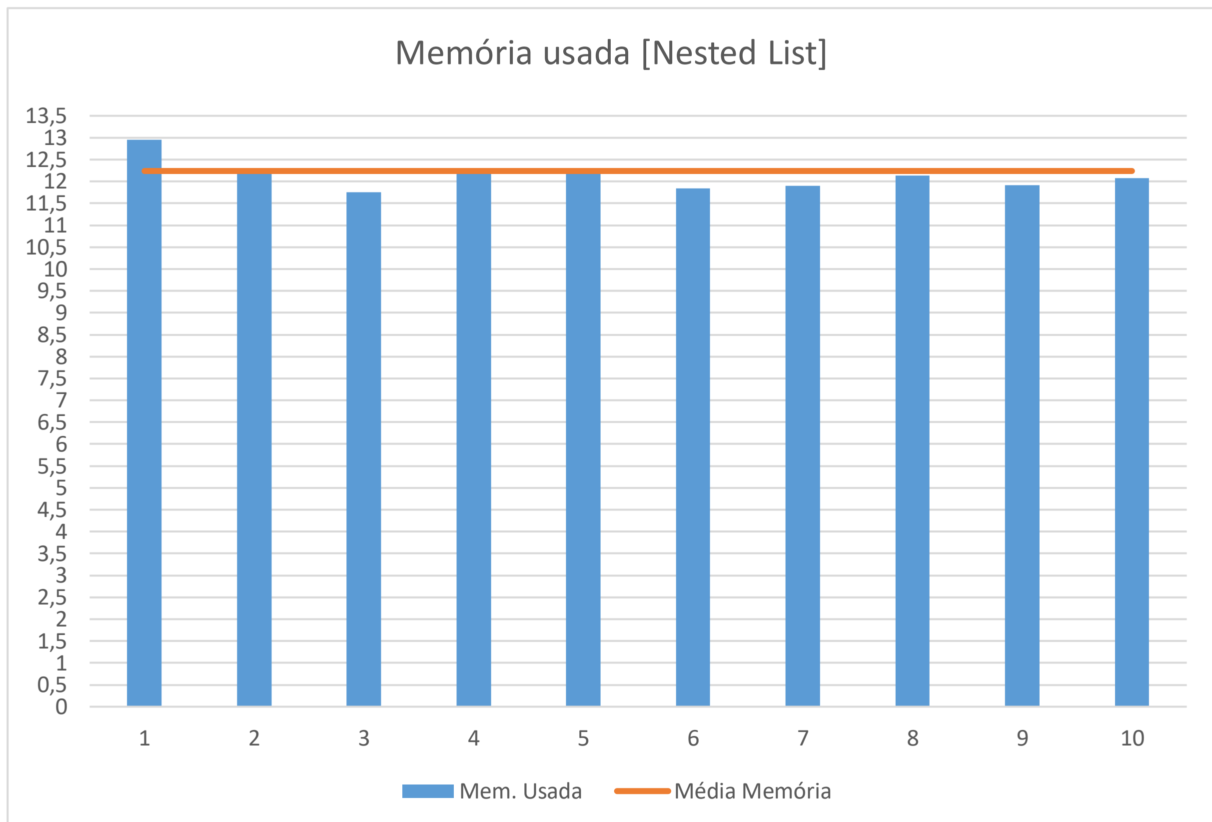

Figura 29 - Gráfico de uso de memória na Implementação Nested List

Comparação das 3 implementações no CPU

Após a realização das execuções, verificamos que a implementação com numpy é sem dúvidas a execução mais rápida. É cerca de 59x mais rápida que a implementação dos Ciclos For e 52x mais rápida que a das NestedLists.

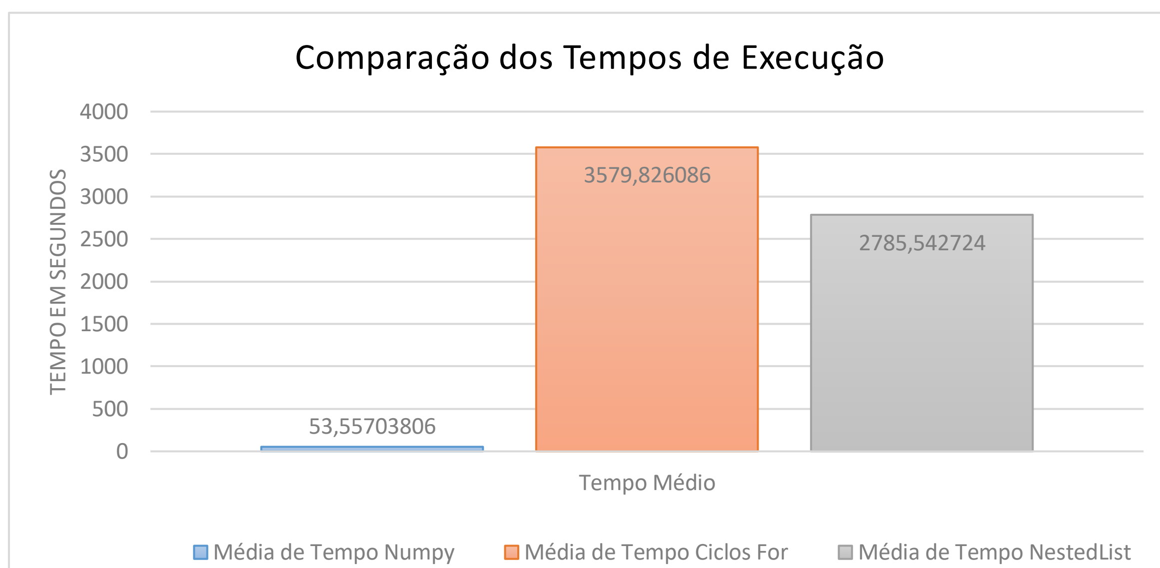

Figura 30 - Comparação da média do tempo de execução das 3 implementações

Quanto ao uso de memória, podemos verificar que, apesar de ser a implementação mais lenta, a implementação do ciclo ***for*** é a que usa menos memória ao longo de toda a execução, enquanto a implementação mais rápida utiliza mais memória.

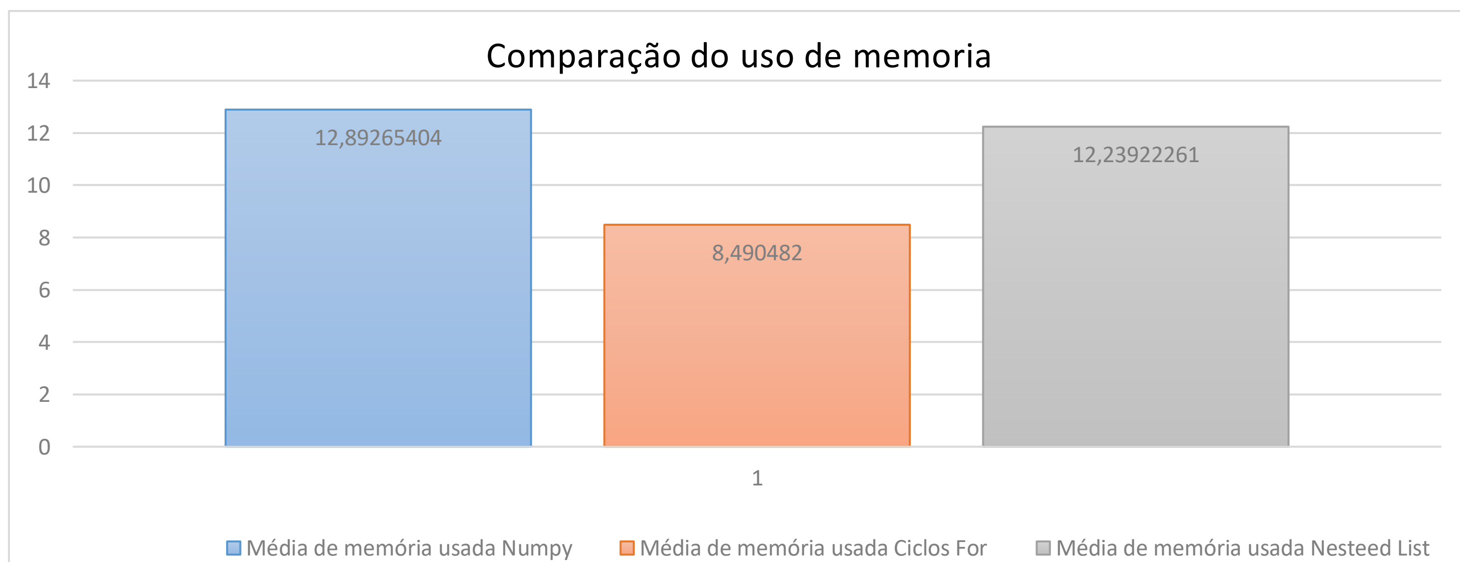

Figura 31 - Comparação da média do uso da memória das 3 implementações

Naive – Cálculo da Matriz por completo

Após correr esta implementação, verificaram-se os seguintes resultados:

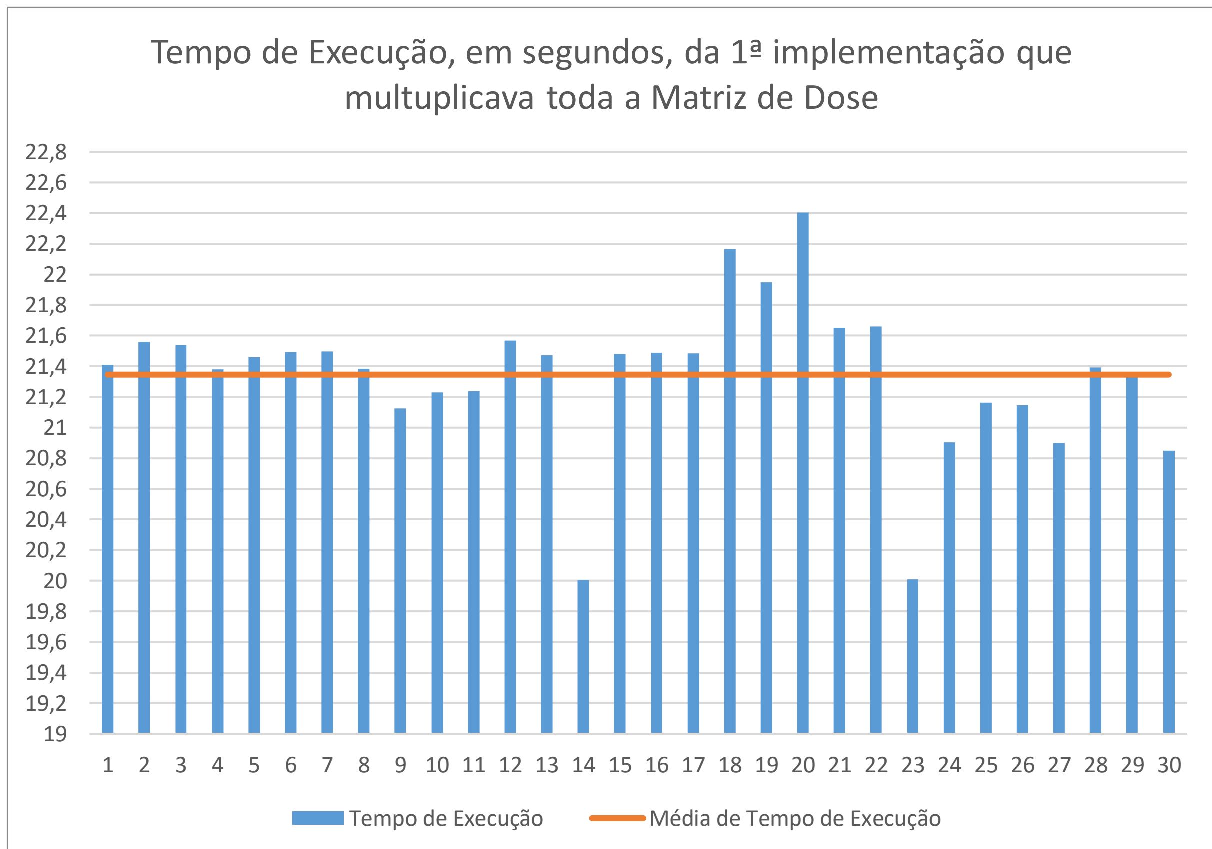

Figura 32 - Gráfico de Tempo de Execução na Implementação Cálculo da matriz por completo

Verificamos que ao longo das 30 execuções, o tempo de execução teve pequenas oscilações, desde 20 segundos a cerca de 23 segundos, onde a média obtida foi de 21,34547 segundos. Com isto, podemos dizer que esta solução é pouco otimizada visto que demora mais que a implementação base em Matlab.

C4.6.2 Resultados Finais

Como foi indicado na secção do Setup, os principais testes foram direcionados para dois scripts, estes que obtiveram os melhores resultados ao longo que várias implementações. Os scripts foram corridos várias vezes para obter uma maior amostra de tempos de execução possível. Após isso, foram comparados os dados e obtiveram-se os seguintes resultados.

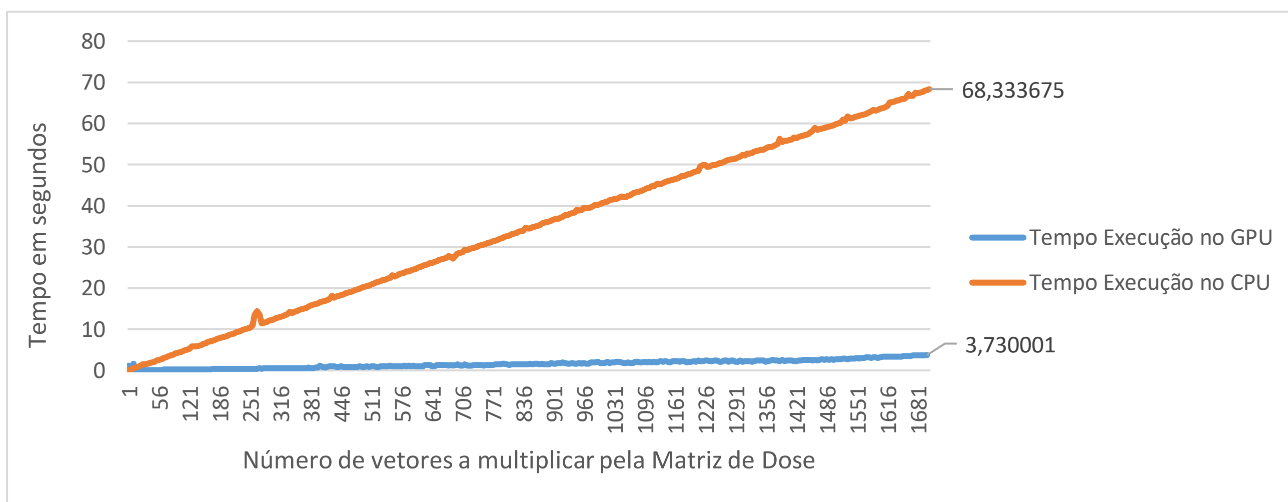

Figura 33 - Comparação entre os tempos de execução do cálculo no CPU e no GPU com Sparse Matrix

O que verificamos com a leitura deste gráfico, é que o cálculo em GPU é muito mais rápido do que em CPU, e é cada vez mais notório a cada aumento do número de vetores a multiplicar.

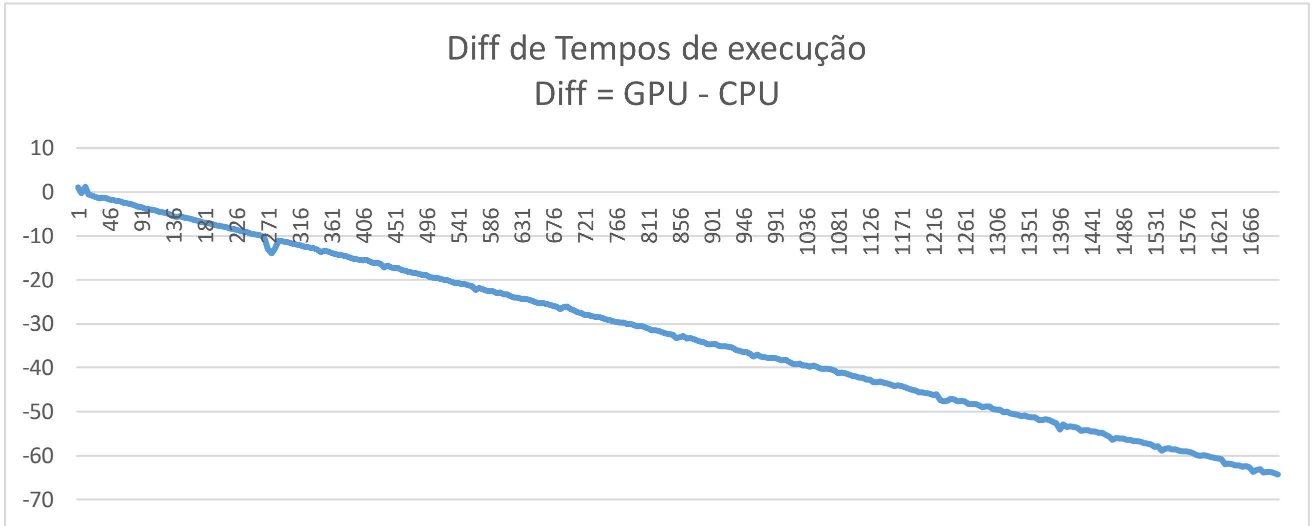

Figura 34 - Diferença entre os tempos de execução do cálculo no CPU e no GPU com Sparse Matrix

Aqui podemos comprovar o aumento da diferença dos tempos de execução das duas diferentes implementações consoante o aumento no número de vetores a multiplicar pela matriz de dose.

4.7 Integração com sistema principal

Após correr as implementações, recolher os resultados e compará-los, a próxima fase seria a integração da melhor implementação com o projeto. Como o módulo que está inserido no projeto está a ser desenvolvido em Matlab, e a nossa implementação em Python com CUDA, tinha de ser feita uma ligação entre os dois.

Deste modo, a primeira forma de tentar de o conseguirmos fazer foi através do Matlab, chamar uma função em python, enviando para ela dois argumentos, que seriam a Matriz de Dose e o vetor a multiplicar por ela, recebendo o resultado das duas. Ao testar, verificou-se que estar resolução não iria funcionar pois o Matlab não suporta a conversão de Sparse Matrix, para Python.

```
tic;
load('HeadAndNeck.mat');
vetor = sparse(double(vetor_x));

pyversion;
pathToSpeech = fileparts(which('multiplyWithMatlab.py'));
if count(py.sys.path, pathToSpeech) == 0
    insert(py.sys.path, int32(0), pathToSpeech);

end

resultado = py.multiplyWithMatlab.calculateMultiplication(dose_matrix, vetor);
toc;
```

Figura 35 - Excerto do Código da integração com Python em Matlab

Com isso, aparecia o seguinte erro na “Command Window” do Matlab:

```
Error using py.multiplyWithMatlab.calculateMultiplication
Conversion of MATLAB 'sparse double' array to Python is not supported.
```

Figura 36 - Erro de o Matlab não suportar conversão de 'sparse double' para Python

Consequentemente, foi necessário arranjar uma solução diferente. A solução passava por no Matlab mandar executar uma função em Python, em que ela ia multiplicar as matrizes e no fim guardar o resultado num ficheiro .mat. Após isso, já no Matlab, faz-se load do ficheiro onde o resultado foi guardado e assim conseguir ter acesso a ele.

```
tic;

if count(py.sys.path, '') == 0
    insert(py.sys.path, int32(0), '');
end

mod = py.importlib.import_module('multiplyWithMatlab');
py.importlib.reload(mod);
py.multiplyWithMatlab.calculateMultiplication;

toc;

load('resultado.mat');
```

Figura 37 - Nova solução de integração com Python

Após todas as implementações e todos os testes, foi ainda produzido um Package com os métodos para o cálculo de multiplicação de matrizes em CPU e GPU e também para ler o ficheiro .mat. Assim, ao invés de termos de andar com ficheiros atrás, é só instalar a Package e chamá-lo no script que for necessário.

```
from __future__ import print_function
import sys
import numpy as np
import scipy.io
from scipy.sparse import csr_matrix
import os, psutil
import openpyxl
import datetime
import cupyx as cpx
import cupy as cp
import random

def multiply_Matrix_GPU(mat_sparse_dose, arrayVetores, NV):

    mat_dose_gpu = cpx.scipy.sparse.csc_matrix(mat_sparse_dose)
    resultados = []

    #for matrix sparse calculation on the gpu
    for i in range(len(arrayVetores)):
        vector_host = np.float64(arrayVetores[i])
        vector_gpu = cp.asarray(vector_host)
        resultado = mat_dose_gpu.dot(vector_gpu)
        resultados.append(resultado)

    return resultados

def multiply_Matrix_CPU(mat_sparse_dose, arrayVetores, NV):

    resultados = []

    #for matrix sparse calculation on the cpu
    for l in range(len(arrayVetores)):
        resultado = mat_sparse_dose.dot(csr_matrix(arrayVetores[l]))
        resultados.append(resultado)

    return resultados


def read_mat(path):
    file = scipy.io.loadmat(path)
    return file
```

Figura 38 - Funções presentes na Package

Capítulo 5
Plano de Trabalho

O plano de trabalho para o segundo semestre será diferente do plano de trabalho do primeiro semestre. O principal aspeto para que o plano de trabalho seja diferente e mais trabalhoso é que, no segundo semestre, vai ser em tempo integral. Com isso, ficou definido o diagrama de Gantt apresentado na Figura 9.

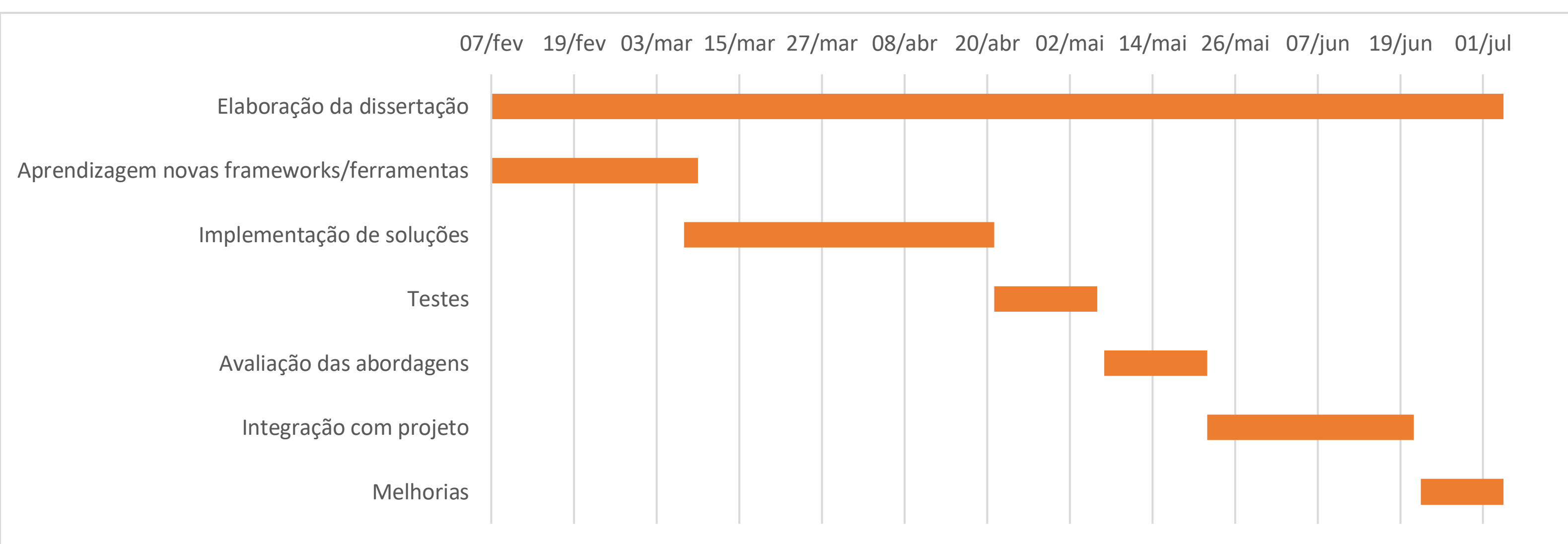

Figura 39 - Plano de trabalho para o segundo semestre com as tarefas a realizar

Colocar dia de cada etapa do plano

- Elaboração de dissertação: ao longo de todo o semestre, a dissertação vai ser um trabalho em progresso, onde se vai acrescentando conteúdo útil e necessário para a realização do mesmo, para que, deste modo, não se deixe todo para o fim e fique uma dissertação mais completa.

- Familiarização com novas frameworks/ferramentas: vai ser uma das tarefas importantes, pois, só conhecendo as novas ferramentas a utilizar, se irá conseguir fazer as implementações das soluções.

- Implementação das soluções: após o estudo das novas ferramentas, vai passar-se para a implementação de várias soluções e como, definido no objetivo principal desta dissertação, é passar as implementações no CPU para GPU.

- Testes: fase para verificar o desempenho e a performance das soluções implementadas.

- Avaliação das abordagens: após os testes, vai-se avaliar as abordagens e concluir qual foi a melhor, através dos resultados obtidos como, por exemplo, o tempo de execução ou a memória ocupada.

- Integração com o Projeto: após a realização dos testes e das avaliações das abordagens, ir-se-á fazer a integração com o projeto da dissertação.

- Melhorias: por fim, uma pequena fase para, se for necessário, realizar alterações ou até repetir uma das tarefas anteriores.

Durante este segundo semestre, o plano de trabalho sofreu algumas alterações. Algumas das tarefas demoraram mais tempo, como, por exemplo, os testes e a avaliação das abordagens, pois houve muitas alterações nas implementações de modo a conseguir ter uns resultados mais eficientes. Já, por exemplo, a integração com o projeto foi uma etapa que, apesar de ter dado algumas implicações, foi a etapa que demorou menos tempo. Como era de esperar, a realização da dissertação foi feita ao longo de todo o semestre. Apesar de algumas etapas terem demorado mais tempo, foram começadas antecipadamente, o que possibilitou que a etapa das melhorias tivesse uma maior duração para o que fosse necessário.

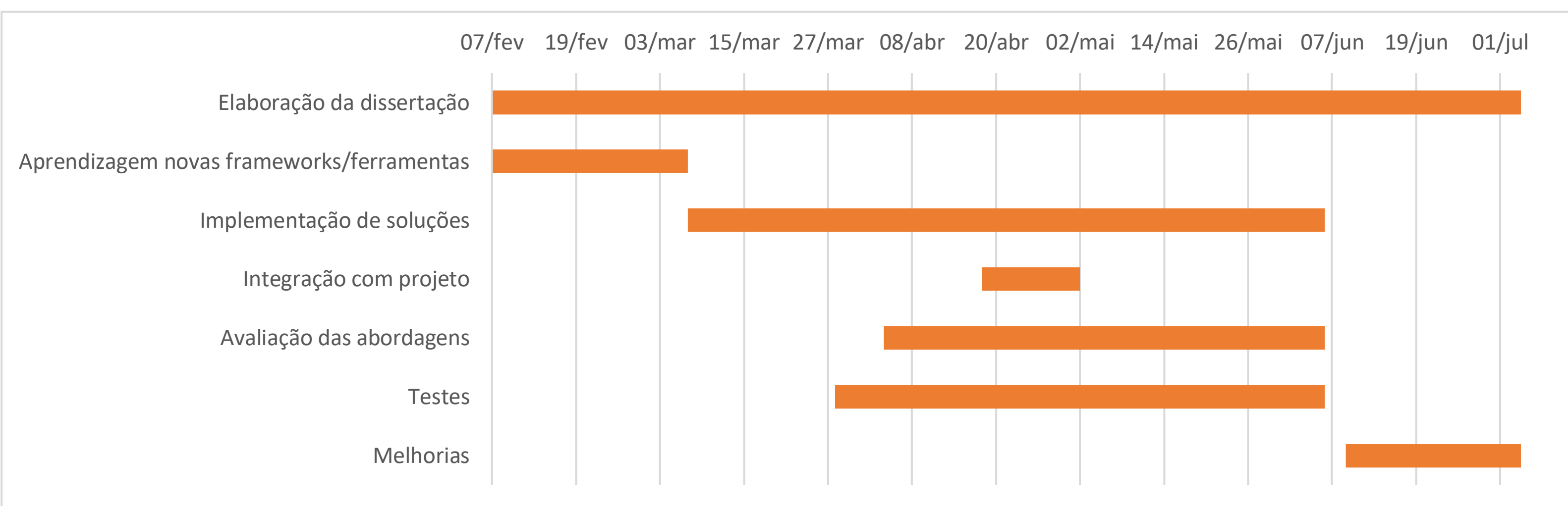

Figura 40 - Plano de trabalho que foi realizado ao longo do segundo semestre

Capítulo 6
Conclusão

Durante o estágio, podemos concluir que foram cumpridos os objetivos propostos inicialmente. Primeiramente, foram feitas diversas pesquisas sobre radioterapia, o que é, como se processa, as técnicas e algoritmos utlizados, e os equipamentos que permitem realizar esta terapia. Após isso, foram feitas investigações sobres API's mais usadas em GPU, de modo a obtermos informação que poderíamos usar no nosso projeto. De seguida, foram realizadas várias e diferentes implementações, tanto no CPU como no GPU. Com essas implementações, foram realizados os devidos testes, tendo em conta o tempo de execução e a memória utilizada durante a execução. Por fim, com os resultados obtidos, concluímos que, com a melhor implementação, em comparação com a execução em Matlab, obtivemos resultados significativamente mais rápidos, o que nos leva a afirmar que o objetivo final de desenvolver uma solução que consiga ter ganho, tanto de performance como de tempo, no cálculo de multiplicação de matrizes, em relação à solução atual, foi alcançado com sucesso.

Em termos de trabalho futuro, o objetivo passará por migrar as implementações para Matlab a fim de tentar otimizar ainda mais as implementações, visto que o projeto está a ser desenvolvido neste software. Desta forma evitar-se-ia a mudança de Matlab pra Python, para executar o cálculo da multiplicação de matrizes e, posteriormente, novamente para o Matlab, para ler os resultados e prosseguir com o processo.

Referências

Abi-Chahla, Fedy (18 de junho de 2008). «Nvidia's CUDA: The End of the CPU?». Tom's Hardware.

Amaro, Hugo Dinis Pereirinha da Silva. Visualization techniques for Big Data. Diss. Universidade de Coimbra, 2015.

Cygler, J., Battista, J. J., Scrimger, J. W., Mah, E., & Antolak, J. (1987). Electron dose distributions in experimental phantoms: a comparison with 2D pencil beam calculations. Physics in Medicine & Biology, 32(9), 1073.

De Martino, Fortuna, et al. "Dose Calculation Algorithms for External Radiation Therapy: An Overview for Practitioners." Applied Sciences 11.15 (2021): 6806.

Duarte, Carlos Pereira. "Implementação de Algoritmos de Reconstrução de Imagens de Tomossíntese" (2016).

Galvin, J. M., Smith, A. R., & Lally, B. (1993). Characterization of a multileaf collimator system. *International Journal of Radiation Oncology* Biology* Physics*, *25*(2), 181-192.

Harris, M. (2007). Optimizing parallel reduction in CUDA. Nvidia developer technology, 2(4), 70.

Hogstrom, K. R., & Almond, P. R. (1983). Comparison of experimental and calculated dose distributions. Electron beam dose planning at the MD Anderson Hospital. Acta radiologica. Supplementum, 364, 89-99.

Hogstrom, K. R., Mills, M. D., & Almond, P. R. (1981). Electron beam dose calculations. Physics in Medicine & Biology, 26(3), 445.

Jia, Xun, et al. "GPU-based fast Monte Carlo simulation for radiotherapy dose calculation". (2015)
Kanematsu, N. "A proton dose calculation code for treatment planning based on the pencil beam algorithm." NIRS publications NIRS-M 122 (1997): 1-20.

Keall, P. J., et al. "Monte Carlo dose calculations for dynamic IMRT treatments." Physics in Medicine & Biology 46.4 (2001): 929.

Kim, Sung Jin, Sung Kyu Kim, and Dong Ho Kim. "Comparison of pencil-beam, collapsed-cone and Monte-Carlo algorithms in radiotherapy treatment planning for 6-MV photons." Journal of the Korean Physical Society 67.1 (2015): 153-158.

Li, J. S., et al. "Validation of a Monte Carlo dose calculation tool for radiotherapy treatment planning." Physics in Medicine & Biology 45.10 (2000): 2969.

Mah, Ernest, et al. "Experimental evaluation of a 2D and 3D electron pencil beam algorithm." Physics in Medicine & Biology 34.9 (1989): 1179.

Michailidis, P. D., & Margaritis, K. G. (2013). Accelerating kernel density estimation on the GPU using the CUDA framework. Applied Mathematical Sciences, 7(30), 1447-1476.

Monteiro, Armanda G. "Comparação de duas Técnicas de Radioterapia de Intensidade Modulada no tratamento do cancro da próstata." (2016).

Nicolucci, Patrícia. "3DCRT e IMRT". (2020).

Saini, Jay & Traneus, Erik & Maes, Dominic & Regmi, Rajesh & Bowen, Stephen & Bloch, Charles & Wong, Tony. (2018). Advanced Proton Beam Dosimetry Part I: Review and performance evaluation of dose calculation algorithms. Translational Lung Cancer Research. 7. 171-179. 10.21037/tlcr.2018.04.05.

Sarrut, David, et al. "A review of the use and potential of the GATE Monte Carlo simulation code for radiation therapy and dosimetry applications." Medical physics 41.6Part1 (2014): 064301.

Soukup, Martin, Matthias Fippel, and Markus Alber. "A pencil beam algorithm for intensity modulated proton therapy derived from Monte Carlo simulations." Physics in Medicine & Biology 50.21 (2005): 5089.

Takahashi, S. (1965). Conformation radiotherapy-rotation techniques as applied to radiography and radiotherapy of cancer. *Acta Radiol*, *242*, 1-142.

Yang, Carl, Aydın Buluç, and John D. Owens. "Design principles for sparse matrix multiplication on the gpu." *European Conference on Parallel Processing*. Springer, Cham, 2018.

Zhi Ouyang. "Univariate Kernel Density Estimation". (2005).