INSTITUTO DE PESQUISAS ENERGÉTICAS E NUCLEARES

AUTARQUIA ASSOCIADA À UNIVERSIDADE DE SÃO PAULO

DESENVOLVIMENTO DE UM SOFTWARE DE MONTE
CARLO PARA TRANSPORTE DE FÓTONS EM
ESTRUTURAS DE VOXELS USANDO UNIDADES DE
PROCESSAMENTO GRÁFICO

MURILLO BELLEZZO

Dissertação apresentada como parte
dos requisitos para obtenção do Grau
de Mestre em Ciências na Área de
Tecnologia Nuclear - Reatores

Orientador: Prof. Dr. Hélio Yoriyaz

São Paulo

2014

Resumo

Sendo o método mais preciso para estimar a dose absorvida em radioterapia, o Método de Monte Carlo (MMC) tem sido amplamente utilizado no planejamento de tratamento radioterápico. No entanto, a sua eficiência pode ser melhorada para aplicações clínicas de rotina. Nesta dissertação é apresentado o código CUBMC, um código de Monte Carlo que simula o transporte de fótons para cálculo de dose, desenvolvido na plataforma CUDA (Compute Unified Device Architecture). A simulação de eventos físicos é baseada no algoritmo presente no código PENELOPE, e as tabelas de seção de choque utilizadas são geradas pela rotina MATERIAL, também presente no código PENELOPE. Os fótons são transportados em objetos simuladores descritos por voxels. Existem duas abordagens distintas utilizadas para a simulação. A primeira delas obriga o fóton a realizar uma parada toda vez que cruza a fronteira de um voxel, a segunda é pelo Método de Woodcock, onde o fóton ignora a existência de fronteiras e é transportado em um meio homogêneo fictício. O código CUBMC tem como objetivo ser uma opção de código simulador que, ao utilizar a capacidade de processamento paralelo de unidades de processamento gráfico (GPU), apresente alto desempenho em máquinas compactas e de baixo custo, podendo assim ser aplicado em casos clínicos e incorporado a sistemas de planejamento de tratamento em radioterapia.

Palavras-chaves: *GPU; dosimetria; simulação de Monte Carlo; radioterapia; transporte de fótons; PENELOPE; MCNP5*

Abstract

As the most accurate method to estimate absorbed dose in radiotherapy, Monte Carlo Method (MCM) has been widely used in radiotherapy treatment planning. Nevertheless, its efficiency can be improved for clinical routine applications. In this master thesis, the CUBMC code is presented, a GPU-based MC photon transport algorithm for dose calculation under the Compute Unified Device Architecture (CUDA) platform. The simulation of physical events is based on the algorithm used in PENELOPE, and the cross section table used is the one generated by the MATERIAL routine, also present in PENELOPE code. Photons are transported in voxel-based geometries with different compositions. There are two distinct approaches used for transport simulation. The first of them forces the photon to stop at every voxel frontier, the second one is the Woodcock method, where the photon ignores the existence of borders and travels in homogeneous fictitious medium. The CUBMC code aims to be an alternative for Monte Carlo simulator code that, by using the capability of parallel processing of graphics processing units (GPU), provides high performance simulations in low cost compact machines, and thus can be applied in clinical cases and incorporated in treatment planning systems for radiotherapy.

Keywords: *GPU; dosimetry; Monte Carlo simulation; radiotherapy; photon transport; PENELOPE; MCNP5*

Sumário

1	Introdução	1
2	Objetivos	5
3	Revisão Bibliográfica	7
4	Materiais e Métodos	10
4.1	Unidade de Processamento Gráfico (GPU)	10
4.1.1	CUDA (Compute Unified Device Architecture)	11
4.1.2	Arquitetura de uma GPU	11
4.1.2.1	Tipos de Memória	13
4.1.3	Ocupação	16
4.2	Método de Monte Carlo	16
4.2.1	Técnicas de amostragem	18
4.2.1.1	Método da Inversão ou Método Direto	18
4.2.1.2	Distribuições Discretas	19
4.2.1.3	Técnica da rejeição	20
4.2.2	Gerador de números pseudoaleatórios	22
4.2.2.1	RANECU	25
4.2.2.2	Paralelização do gerador de números pseudoaleatórios . . .	27
4.2.3	Transporte de fótons	28
4.2.3.1	Sorteio do passo	29
4.2.3.2	Propagação da partícula	31

4.2.3.3	Sorteio da interação	34
4.2.3.4	Cálculo dos tallies	39
4.3	AMIGOBrachy	40
5	Resultados	42
5.1	Gerador de Números Pseudoaleatórios (PRNG)	42
5.1.1	Comparativo de desempenho entre o RANECU e sua versão paralelizada	43
5.1.2	Função <i>Rand</i>	45
5.2	Cálculo de π	46
5.3	CUBMC	51
5.3.1	Aspectos Gerais	51
5.3.2	Geometria	52
5.3.3	Tabelas de Seção de choque	54
5.3.4	Modelos de fonte de radiação	56
5.3.5	Sorteio do passo	57
5.3.6	Fronteiras	60
5.3.7	Interações	63
5.3.8	Estimadores (Tallies)	64
5.3.9	Finalização da história da partícula	65
5.3.10	Método de Woodcock	65
5.3.11	Simulações	66
6	Discussões	74
7	Conclusão	76
	Referências Bibliográficas	79

1 Introdução

O câncer, em suas diversas formas, é a segunda maior causa de morte no Brasil. Estima-se que em 2014 teremos cerca de 576 mil novos casos. Dentre os diversos tipos de tratamento possíveis, podemos destacar a radioterapia, que consiste em se bombardear células tumorais com uma alta dose de radiação com o intuito de que o tumor regrida [1] [2]. Embora seja um dos métodos mais eficientes no tratamento do câncer, a incidência de radiação nos tecidos sadios tem efeito deletério e deve ser minimizada. Para otimizar a distribuição de dose durante um tratamento, existem softwares de planejamento de tratamento radioterápico 3D que, baseados em imagens de Tomografia Computadorizada (CT), Ressonância Magnética (MR) e Ultrassonografia (US), permitem realizar uma simulação computacional do tratamento para o modelo anatômico específico do paciente [3]. O Método de Monte Carlo (MMC) é um dos melhores métodos para se calcular transporte de radiação e amplamente empregado para realizar essas simulações. No entanto, por se tratar de um método estatístico, é custoso computacionalmente, exigindo computadores potentes para que sua aplicação em sistemas de planejamento seja aplicável em casos clínicos.

O Método de Monte Carlo compreende uma classe de métodos numéricos baseada no uso de números aleatórios que pode ser aplicada para solucionar qualquer problema descrito por uma sucessão de processos estocásticos. Recebeu esse nome na década de 1940 por cientistas integrantes do Projeto Manhattan, em Los Alamos, devido à aleatoriedade em suas simulações, também presente nos jogos de azar, e por causa do bairro de Monte Carlo, pertencente ao principado de Mônaco, ser conhecido na época como a capital mundial dos jogos de azar. Em muitas aplicações, ao se empregar o Método de Monte Carlo, o processo físico pode ser simulado sem a necessidade de se solucionar as equações matemáticas que o descrevem, sendo apenas necessário que ele seja representado por funções de densidade de probabilidade (PDFs). Dessa forma, o método vem sendo amplamente usado para solucionar problemas complexos de física e matemática sendo, muitas vezes, o único meio viável para problemas de múltiplas variáveis independentes para os quais os métodos numéricos convencionais exigiriam um tempo computacional

e/ou quantidade de memória demasiadamente elevados.

Enquanto a solução de um problema por métodos numéricos convencionais consiste na solução de equações algébricas e diferenciais, um problema simulado pelo Método de Monte Carlo tem sua física embutida nas PDFs, que definem a probabilidade da ocorrência dos fenômenos microscópicos, dessa forma, a execução da simulação resume-se ao uso de técnicas de amostragem para se simular de forma determinística uma sequência específica de eventos que passarão a compor um grupo estatístico, do qual obtemos valores médios de grandezas macroscópicas, juntamente com seu erro estatístico. Tais simulações são independentes entre si, característica que permite que o Método de Monte Carlo seja altamente paralelizável.

Dentre os programas computacionais existentes baseados no Método de Monte Carlo destacam-se o MCNP6 [4], PENELOPE [5] e GEANT4 [6]. Estes programas foram desenvolvidos e aperfeiçoados ao longo de décadas de pesquisas na área e hoje são amplamente usados na área de física médica. Entretanto, são ferramentas de uso geral, muito complexas e de difícil adaptação a novas plataformas, exigindo uma demanda de estudos e experiência em simulação por parte dos usuários. Além disso, por serem códigos que possibilitam o transporte de diversas partículas simulando suas principais interações, sua execução pode ser computacionalmente custosa. Esforços vem sendo realizados para a adaptação destes códigos ao processamento paralelo como, por exemplo, no caso do programa PENELOPE [7]. Outros programas como o MCNP já foram adaptados com sucesso para serem executados paralelamente, seja em computadores com processadores de múltiplos núcleos ou em *clusters*, que são supercomputadores compostos por centenas ou milhares de processadores que se comunicam de modo a realizar processamento paralelo.

Uma alternativa aos *clusters* são as unidades de processamento gráfico (GPUs), desenvolvidas originalmente como um processador auxiliar destinado ao processamento gráfico do computador, possuem grande capacidade de processamento lógico aritmético e uma arquitetura otimizada para o paralelismo. Tendo sua evolução alavancada pela indústria de jogos, a capacidade de processamento aritmético das GPUs ultrapassou a das unidades de processamento central (CPUs) e softwares mais recentes fazem uso dessa capacidade de processamento para ganho de desempenho em seus processos, como pode-

mos observar na figura 1.1.

Figura 1.1: Comparativo entre operações de pontos flutuantes por segundo entre as principais CPUs e GPUs

As GPUs não substituem o processador central de um computador, no entanto, por serem processadores mais simples que as CPUs, oferecem um grande poder de processamento aritmético a baixo custo. Ao se comparar GPUs e CPUs de custo semelhante, a GPU apresenta uma capacidade de processamento aritmético dezenas ou até centenas de vezes superior. Visando tirar proveito dessa alta capacidade de processamento, surgiram novos códigos de programação capazes de direcionar parte do fluxo de suas operações para as GPUs. Dentre eles, se destaca o CUDA (Compute Unified Device Architecture), linguagem desenvolvida pela Nvidia para as GPUs da própria empresa, CUDA é baseada nas linguagens C/C++ [8], fato que permite uma rápida curva de aprendizado para programadores já experientes nessas linguagens.

O presente trabalho consiste no desenvolvimento do código CUBMC (CUDA

Based Monte Carlo), um software desenvolvido em CUDA destinado à simulação do transporte de fótons pelo Método de Monte Carlo.

2 Objetivos

O objetivo deste trabalho é o desenvolvimento do software CUBMC (CUDA Based Monte Carlo), um código de Monte Carlo destinado ao transporte de fótons em objetos simuladores descritos por voxels, para que seja uma opção de código simulador que, ao utilizar a capacidade de processamento paralelo de unidades de processamento gráfico (GPUs), apresente alto desempenho em máquinas compactas e de baixo custo, podendo assim ser aplicado em casos clínicos e incorporado a sistemas de planejamento de tratamento de radioterapia.

Dessa forma, a opção de trabalhar com objetos simuladores definidos por voxels busca facilitar a descrição de modelos anatômicos provenientes de imagens de Tomografia Computadorizada (CT), Ressonância Magnética (MR) ou Ultrassonografia (US), que são inicialmente bidimensionais, mas facilmente extrapoladas para voxels. Com o intuito de obter ganho de desempenho, o código CUBMC destina-se exclusivamente à simulação do transporte de fótons, as partículas secundárias geradas nas interações não são transportadas e sua energia é depositada localmente.

A física envolvendo a simulação do transporte da radiação presente no código foi simplificada ao máximo dentro do limite em que ainda apresente um resultado adequado no intervalo energético de interesse para os tratamentos de radioterapia e braquiterapia (1keV - 9MeV). Dentre as interações simuladas (Efeito Fotoelétrico, Espalhamento Rayleigh, Espalhamento Compton e Produção de Pares), apenas o Espalhamento Compton apresenta um algoritmo de simulação mais abrangente, tendo sido extraído sem alterações do código PENELOPE (como o PENELOPE é um código escrito em fortran [9] e CUDA é baseada em C/C++, foi preciso realizar adaptações entre as diferentes linguagens, no entanto, sem alterar a lógica do algoritmo). As outras interações são extremamente simplificadas e serão implementadas à medida que o código CUBMC for aprimorado. As tabelas de seção de choque são geradas pela rotina MATERIAL, também pertencente ao código PENELOPE.

O código CUBMC permite que o transporte dos fótons seja simulado segundo

dois métodos. Um método em que a partícula é obrigada a realizar uma parada toda vez que encontrar uma fronteira da geometria (chamado neste trabalho de ‘método de paradas’), e um método em que a partícula é transportada em um meio homogêneo fictício, ignorando as fronteiras da geometria (Método de Woodcock ou ‘*delta tracking*’). Futuramente, será implementada a possibilidade de se trabalhar com um método misto, que aplica um ou outro método dependendo da região da geometria onde o fóton esteja sendo transportado, conforme implementado no código Serpent [10], um código de Monte Carlo desenvolvido para a simulação de transporte em reatores nucleares.

3 Revisão Bibliográfica

As simulações realizadas pelo método de Monte Carlo tem sido amplamente usadas na área de física médica nos últimos 60 anos. Se pesquisarmos pelo termo ‘Monte Carlo’ na PubMed (<http://www.ncbi.nlm.nih.gov/pubmed>) em fevereiro de 2014, teremos mais de 36 mil retornos, sendo que o primeiro artigo data de setembro de 1949 [11]. Entre os anos de 1967 e 2000, o número de publicações contendo o termo ‘Monte Carlo’ em seu título ou resumo presentes nas revistas *Physics in Medicine and Biology* (PMB) e *Medical Physics* dobraram a cada, aproximadamente, 5 anos [12], conforme ilustra a figura 3.1.

Figura 3.1: Número de publicações contendo o termo ‘Monte Carlo’ em seu título ou resumo presentes nas revistas *Physics in Medicine and Biology* (PMB) e *Medical Physics* ao longo dos anos [12]

Esse crescimento pode ser atribuído ao aumento da capacidade de processamento e redução de custo dos computadores, que alavancou o desenvolvimento de softwares envolvendo o Método de Monte Carlo para simular o transporte de radiação. O emprego destes softwares pode ser destacado para o transporte de radiação em cálculos dosimétricos

de braquiterapia e radioterapia, sendo parte integrante fundamental em alguns sistemas de planejamento de tratamento [13].

Para que os tempos de simulação dos códigos de Monte Carlo fossem viáveis para aplicação clínica em sistemas de planejamento de tratamento, foram desenvolvidos métodos para que esses códigos fossem executados mais rapidamente. Dentre eles, podemos destacar o VMC++ (Voxel Monte Carlo) [14] que é semelhante ao EGSnrc [15], e resultou em simulações ao mínimo uma ordem de grandeza mais rápidas [12]. Também temos o DPM [16], que teve os modelos físicos para as simulações simplificados para que a execução fosse mais rápida, mas ainda obtivesse resultados corretos quando aplicado a sistemas de planejamento radioterápicos e o MCDOSE [17]. Ainda assim, nenhum deles obteve a mesma eficiência que o código VMC++.

A descrição da geometria por voxels mostrou-se adequada para objetos simuladores provenientes de modelos anatômicos, uma vez que imagens de tomografia computadorizada (CT) e ressonância magnética (RM) são facilmente exportadas para modelos tridimensionais voxelizados, além de permitir descrições mais detalhadas do corpo humano [18] [19] e específicas para cada paciente [20].

Outra forma de se acelerar uma simulação de Monte Carlo é paralelizar o processo. Uma vez que as partículas simuladas são independentes entre si, elas podem ser simuladas simultaneamente por núcleos processadores distintos. Esforços foram feitos para paralelizar códigos de Monte Carlo já existentes, como foi feito com o PENELOPE [21], ou o desenvolvimento de novas versões dos códigos englobando a possibilidade de processamento paralelo como uma de suas atualizações, como o MCNP5 [22].

A execução em paralelo possibilitou que códigos de Monte Carlo mais abrangentes fossem executados em tempos comparáveis aos de códigos simplificados. No entanto, existia a limitação financeira devido ao alto custo de computadores com grande capacidade de processamento paralelo.

Com o advento do uso das Unidades de Processamento Gráfico (GPUs) (GPUs apresentam uma elevada capacidade de processamento paralelo) como processadores auxiliares, o desenvolvimento ou adaptação de softwares de Monte Carlo otimizados para processamento paralelo foi alavancado [23].

Códigos como o PENELOPE [24] e DPM [25] foram migrados para a execução em GPU, contudo, os ganhos de desempenho foram modestos quando comparados ao potencial de ganho. Isso devido, principalmente, a serialização que ocorre ao se gerar partículas secundárias nesses códigos. Já o código GPUMCD [26] obteve ganho de desempenho de 200x frente ao DPM após corrigir o tratamento das partículas secundárias. Para simulações que consideram apenas o transporte dos fótons, ganhos de desempenho de até 1000x foram obtidos para geometrias simples [27].

Em 2011, o código gCTB [28], desenvolvido sob a plataforma CUDA [29], obteve ganhos de 76.6x frente ao código EGSnrc para simulações para o modelo anatômico de cabeça Zubal [30], e de 400x para uma geometria composta por um meio homogêneo de água.

Embora o desenvolvimento de códigos de Monte Carlo otimizados para execução em GPU ainda seja incipiente, ganhos expressivos já foram obtidos. O poder de processamento paralelo das GPUs ainda é crescente, e seu baixo custo faz com que exista um grande potencial de que elas venham a substituir os *clusters* para simulações de Monte Carlo de alta performance.

4 Materiais e Métodos

4.1 Unidade de Processamento Gráfico (GPU)

A GPU surgiu como um processador auxiliar destinado para processamento gráfico, com o intuito de reduzir a quantidade de processamento lógico aritmético realizado pela unidade de processamento central (CPU).

Analisando a arquitetura de uma CPU e de uma GPU (fig. 4.1), podemos observar que uma CPU tem a sua área de controle desenvolvida enquanto suas unidades lógicas aritméticas (ALU) são escassas. Já a GPU tem apenas alguns poucos núcleos de controle e muitas unidades ALU.

Figura 4.1: Diferença arquitetônica entre CPU e GPU

Tal diferença na arquitetura faz com que os processadores centrais sejam dispositivos otimizados para o gerenciamento paralelo de tarefas mas pouco eficientes para realizar operações aritméticas. Já as unidades de processamento gráfico possuem pouca capacidade no que tange ao processamento de múltiplas tarefas, mas são altamente eficientes na execução de operações aritméticas.

Visando tirar proveito dessa alta capacidade de processamento aritmético paralelo das GPUs, surgiram novos códigos de programação capazes de direcionar parte do fluxo de suas operações para as GPUs. Dentre eles, se destaca o CUDA (Compute Unified Device Architecture), que é a linguagem usada nesse trabalho.

4.1.1 CUDA (Compute Unified Device Architecture)

CUDA é uma linguagem de programação desenvolvida pela NVIDIA voltada para as GPUs da própria empresa. Visando uma rápida curva de aprendizagem para programadores já familiarizados com outros códigos, CUDA é uma plataforma com versões que compreendem os comandos e as bibliotecas existentes em C, C++, C# e Fortran. [29]

CUDA C é uma linguagem semelhante à linguagem C/C++, com o diferencial de possuir ferramentas com as quais é possível estabelecer uma comunicação entre a GPU e a CPU. Dessa forma, as partes do código que são responsáveis por operações aritméticas podem ser executadas na GPU.

Entretanto, para um desempenho otimizado, o código deve ter seus parâmetros (discutidos em detalhes mais adiante) ajustados segundo a capacidade da GPU na qual será executado. Diferentes gerações de GPUs NVIDIA apresentam diferentes compatibilidades CUDA. Essas distinções fazem com que a forma como a GPU gerencia os processos em paralelo, e a memória disponível para cada processo seja diferente.

4.1.2 Arquitetura de uma GPU

Embora possuam a mesma estrutura geral, a arquitetura das unidades de processamento gráfico compatíveis com a tecnologia CUDA varia conforme a geração das GPUs.

Cada GPU compatível com a tecnologia CUDA é dividida fisicamente por um grupo de multiprocessadores compostos por múltiplos núcleos, de modo que, uma GPU tenha de centenas à milhares de núcleos CUDA. Essa divisão é intrínseca de cada GPU e não pode ser alterada pelo código.

A GPU executa tarefas (*kernels*) através de *warps* compostos por 32 *threads* (menor unidade de processamento de uma GPU). Todos os *threads* realizarão os mesmos processos, mas com valores de entrada distintos. Caso os parâmetros de execução não ocupem todos os *threads*, parte do *warp* ficará ociosa.

Para se ter um exemplo na figura 4.2 temos o esquema da GPU GTX 560 Ti (usada nesse trabalho). Ela possui 8 multiprocessadores de 48 núcleos cada, totalizando

384 núcleos. Cada núcleo pode executar um warp (de 32 *threads*) por vez. Assim, a ocupação máxima da GPU é de 12288 *threads*.

Figura 4.2: Modelo esquemático da arquitetura de uma GPU GTX 560 Ti (à esquerda) e de um de seus multiprocessadores (à direita)

Além da divisão física presente nas GPUs, existem parâmetros virtuais que definem uma hierarquia referente à execução de um *kernel* enviado pela CPU (host).

A menor unidade de processamento ao se executar um *kernel* é o *thread*. Os *threads* não são núcleos físicos, são organizados em blocos, e são responsáveis por realizar as operações aritméticas de uma determinada função. Sua organização pode ser unidimensional, bidimensional e até tridimensional. Diferente formas de agrupamento podem facilitar a identificação dos *threads* para um determinado problema. Agrupamentos bidimensionais, por exemplo, podem ser interessantes ao se trabalhar com matrizes, no entanto, a organização unidimensional tende a promover um acesso à memória de maneira mais eficiente. Todos os *threads* de um mesmo bloco tem acesso a uma memória comum (memória compartilhada ou ‘shared’) e devem ser executadas pelo mesmo *kernel*.

Os blocos são organizados em grades (figura 4.3) sendo que, os blocos de uma mesma grade também são executados pelo mesmo *kernel*, mas são independentes entre si

no que diz respeito à memória e não são executados em sincronia. Desse modo, os *threads* de um bloco não cooperam com *threads* de outro bloco.

Cada núcleo CUDA será responsável pela execução de um bloco. Sendo assim, existe um limite para o número de blocos que um multiprocessador pode executar simultaneamente. Caso o número de blocos a serem executados exceda o número de núcleos existentes na GPU, é criada uma fila e os blocos serão executados à medida que outros blocos encerrem suas atividades.

Figura 4.3: Esquema das divisões hierárquicas na execução de um *kernel*

4.1.2.1 Tipos de Memória

Para um melhor aproveitamento da massiva capacidade de paralelização das unidades de processamento gráfico é necessário usar de forma adequada os diferentes tipos de memória nelas presentes, pois cada tipo de memória apresenta características diferentes, como forma de escrita e leitura, capacidade de armazenamento, tempo necessário para a leitura e escrita e otimização para acesso simultâneo.

A figura 4.4 ilustra onde cada tipo de memória fica alocada fisicamente na GPU e quem pode acessá-la.

Figura 4.4: Modelo esquemático das diferentes memórias presentes na GPU

- Memória Global

A memória global é a única memória que pode ser acessada para leitura e escrita, tanto pela CPU (host) quanto pela GPU (device). Desse modo, torna-se a principal responsável pela troca de dados entre esses processadores. Apresenta a maior capacidade de alocação da GPU (Gb), e a informação nela contida persiste durante a execução inteira do programa, podendo ser acessada por todos os elementos da grade. No entanto, não é otimizada para acessos simultâneos e exige um tempo longo para ser lida (~ 100 -150 ciclos).

A memória global é a memória usada como padrão para a comunicação entre CPU e GPU caso o código não especifique o uso de outra memória.

- Memória Constante

Semelhante à memória global, pode ser lida e escrita pelo host mas a GPU só tem permissão para leitura. Assim como na memória global, a informação contida na memória constante persiste durante a execução inteira do aplicativo. Dessa forma, é utilizada para o armazenamento de informações que não serão alteradas durante a execução do *kernel*, podendo ser acessada por todos os elementos da grade. É otimizada para acessos simultâneos e possui um tempo de leitura moderado (~ 1 -100 ciclos), contudo, apresenta capacidade limitada de armazenamento (64Kb/multiprocessador).

- Memória de Registro

Memória usada automaticamente pelo *thread* para alocação de variáveis locais. Possui o menor tempo de acesso (~ 1 ciclo) mas é visível apenas pelo *thread*. Cada multiprocessador tem uma quantidade fixa de memória de registro, que é dividida entre os *threads* nele presentes. Tem a duração da execução do *kernel*.

Um maior uso da memória de registro (acarretando em menor uso das demais memórias) pode resultar em uma significativa redução no tempo de execução do código.

- Memória Compartilhada (Shared)

Semelhante à memória de registro, mas pode ser acessada por todos os *threads* de um mesmo bloco. Embora tenha acesso mais lento que o da memória de registro (de 3 a 6 vezes mais lento), é especialmente útil para a comunicação entre os *threads* sem que seja necessário acessar a memória global (muito mais lenta).

- Memória de Textura

A memória de textura é a própria memória global, no entanto, é acessada por um cache dedicado apenas à leitura e otimizado para um acesso mais rápido de elementos vizinhos (no sistema de coordenadas definido na memória de textura).

A memória de textura é uma ótima opção para se armazenar as informações sobre o material, por exemplo, uma vez que voxels vizinhos tendem a ser acessados com maior frequência durante a história de uma partícula.

4.1.3 Ocupação

Ocupação é definida como a razão entre o número de warps sendo processados e o número máximo de warps permitido pela GPU.

A princípio, uma maior ocupação significaria um maior uso dos recursos da GPU. Entretanto, isso não implica necessariamente em um melhor desempenho. Cada multiprocessador possui um limite para memória de registro e para memória compartilhada (que são as memórias de acesso mais rápido da GPU). Sendo assim, quanto menos *threads* forem executados simultaneamente por processador, maior a quantidade de memória disponível para cada um deles.

Sacrificar o grau de paralelização em detrimento de mais registros disponíveis por *thread* pode ser lucrativo em relação à eficiência do código devido ao acesso à memória de registro ser muito mais rápido que o acesso à memória global.

Contudo, também existe um limite máximo de memória por *thread*. Tais peculiaridades fazem com que existam planilhas para o cálculo da ocupação da GPU dada uma configuração. No entanto, testes empíricos acabam sendo necessários para se obter a configuração de melhor desempenho para um determinado problema.

4.2 Método de Monte Carlo

O Método de Monte Carlo (MMC) compreende uma classe de métodos numéricos baseados no uso de números aleatórios. Recebeu esse nome na década de 1940 por cientistas integrantes do Projeto Manhattan, em Los Alamos, devido à aleatoriedade em suas simulações estatísticas, também presente nos jogos de azar, e por causa do bairro de Monte Carlo, pertencente ao principado de Mônaco, ser conhecido na época como a capital mundial dos jogos de azar.

Desde então, o método vem sendo amplamente usado para solucionar problemas complexos de física e matemática sendo, muitas vezes, o único meio viável para problemas de múltiplas variáveis independentes para os quais os métodos numéricos convencionais exigiriam um tempo computacional, e/ou quantidade de memória demasiadamente elevados.

O Método de Monte Carlo é um método estatístico que pode ser aplicado para solucionar qualquer problema cuja representação possa ser descrita por uma sucessão de processos estocásticos. Em muitas aplicações o processo físico pode ser simulado sem a necessidade de se solucionar as equações matemáticas que o descrevem, sendo apenas necessário que ele seja representado por funções de densidade de probabilidade (PDFs).

A amostragem aleatória dessas PDFs simula de forma determinística uma sequência específica de eventos e o resultado dessa simulação passa a compor um grupo estatístico. Desse grupo estatístico, obtemos valores médios das grandezas macroscópicas de interesse, juntamente com seu erro estatístico. Dessa forma, temos a solução de sistemas macroscópicos através de simulações das interações microscópicas.

Por se tratar de um método estatístico, é necessário que as simulações (também chamadas de histórias) sejam realizadas por diversas vezes até que o erro estatístico (desvio padrão) atinja a precisão desejada.

Ou seja, enquanto a solução de um problema por métodos numéricos convencionais consiste na solução de equações algébricas e diferenciais, um problema simulado pelo Método de Monte Carlo tem sua física embutida nas PDFs, que definem a probabilidade da ocorrência dos fenômenos microscópicos, dessa forma, a execução da simulação resume-se ao uso de técnicas de amostragem para se chegar à solução desejada.

Ao se simular o transporte de radiação, a história de uma partícula é descrita como uma sucessão de passos aleatórios seguidos de interações. Dependendo da interação sofrida pela partícula sua direção de propagação pode mudar, parte de sua energia pode ser depositada no meio e partículas secundárias podem ser geradas. A história de uma partícula é simulada até o momento em que ela abandone a geometria do problema, atinja uma energia mínima (energia de corte) ou seja absorvida.

A história simulada de uma partícula não corresponde necessariamente ao comportamento de nenhuma partícula real, no entanto, seu comportamento é regido por distribuições estatísticas baseadas em modelos físicos. Desse modo, para um número grande o bastante de histórias, o resultado macroscópico obtido por análise estatística converge para o resultado correspondente à solução determinística da equação de Boltzman (que descreve o transporte de radiação).

Embora o Método de Monte Carlo seja de mais fácil implementação que a solução da equação de Boltzman para problemas de geometria complexa, devido a sua natureza aleatória, o resultado obtido sempre terá um erro estatístico associado. Para reduzir esse erro, é necessário realizar simulações com um maior número de histórias, e isso implica em um maior custo computacional.

4.2.1 Técnicas de amostragem

A amostragem correta de uma PDF é fundamental para que a simulação realizada pelo Método de Monte Carlo seja fiel ao processo representado pela PDF. Para garantir o caráter randômico da amostragem, um código de Monte Carlo emprega o uso de números racionais aleatórios¹ ξ uniformemente distribuídos entre 0 e 1.

4.2.1.1 Método da Inversão ou Método Direto

Seja $f(x)$ a função que descreve a densidade de probabilidade de ocorrência de um determinado fenômeno em um intervalo $[a : b]$, sua função de distribuição acumulativa (CDF) é definida como sendo:

$$F(x') = \int_a^{x'} f(x) dx \quad (4.1)$$

Sendo $f(x)$ normalizada no intervalo $[a : b]$, temos que $F(x)$ é uma função contínua, crescente, com $F(a) = 0$ e $F(b) = 1$ que possui função inversa. Dessa forma, se temos $F(x) = \xi$, também temos uma função inversa $F^{-1}(\xi) = x$, que define um modo direto de se relacionar as variáveis x e ξ .

Para a variável ξ variando uniformemente no intervalo $[0 : 1]$ a probabilidade de se obter um determinado valor de x é igual a $f(x)$.

- Exemplos:

Distribuição uniforme

Tomemos como exemplo inicial o caso de uma distribuição uniforme já normalizada no intervalo $[a : b]$:

¹Na prática, usa-se números pseudoaleatórios. Esse tema será abordado na seção 4.2.2, página 22

$$f(x) = \frac{1}{b-a}$$

Obtemos a CDF através da integração de $f(x)$ no intervalo $[a : x']$:

$$F(x') = \int_a^{x'} \frac{1}{b-a} dx = \frac{x' - a}{b-a} = \xi \quad (4.2)$$

Isolando x temos a função inversa:

$$x = a + \xi(b-a) = F^{-1}(\xi) \quad (4.3)$$

Função exponencialmente decrescente

Para o caso de uma função exponencialmente decrescente, temos a distribuição normalizada no intervalo $[0 : \infty[$ e obtemos a função inversa pelo mesmo método:

$$f(x) = \Sigma_T e^{-\Sigma_T x} \quad (4.4)$$

$$F(x') = \int_0^{x'} \Sigma_T e^{-\Sigma_T x} dx = 1 - e^{-\Sigma_T x} = \xi \quad (4.5)$$

$$x = \frac{-\ln(1-\xi)}{\Sigma_T} = F^{-1}(\xi) \quad (4.6)$$

Embora a ideia de se calcular a função inversa seja conceitualmente simples, PDFs mais complexas podem fazer com que seu cálculo seja de difícil execução.

4.2.1.2 Distribuições Discretas

O método da inversão também pode ser aplicado para distribuições discretas. Suponha que a variável x pode assumir qualquer valor inteiro entre 1 e N , com as respectivas probabilidades $p_1, p_2, \dots, p_N$ normalizadas.

Desse modo, sua função cumulativa é dada por:

$$F(x) = \begin{cases} 0 \rightarrow x < 1 \\ \sum_{i=1}^x p_i \rightarrow 1 \leq x \leq N \\ 1 \rightarrow x > N \end{cases} \quad (4.7)$$

Novamente, para um número aleatório ξ , de forma que $F(x_i) < \xi < F(x_{i+1})$, o valor sorteado é x_i .

Tomemos como exemplo o lançamento de um dado de 6 faces. Sendo x o valor da face, ele tem igual probabilidade de assumir os valores 1, 2, 3, 4, 5 e 6. Dessa forma, sua PDF é definida por $f(x) = 1/6$, para $x = 1, 2, \dots, 6$.

A figura 4.5 representa $f(x)$.

Figura 4.5: PDF para o lançamento de um dado de 6 faces

Sendo assim, sua função cumulativa é definida por $F(x) = \sum_{i=1}^x 1/6$, representada pela figura 4.6.

Para um número aleatório ξ , obtemos o valor referente de x .

O método discreto é amplamente aplicado em leitura de tabelas, no entanto, por depender do resultado de somatórias, pode ser mais lento que o método direto.

4.2.1.3 Técnica da rejeição

O método da rejeição é um dos métodos mais simples de se amostrar uma PDF. No entanto, embora seja simples, apresenta uma eficiência menor do que dos outros métodos.

O método consiste de duas etapas distintas. Na primeira, calcula-se o valor de $f(x)$ para um valor de x aleatório variando uniformemente no intervalo a ser amostrado

Figura 4.6: Função acumulativa para um lançamento de um dado de 6 faces

da PDF.

Logo, se $f(x)$ é definida no intervalo $[a : b]$, calculamos $f(x)$ para $x = a + \xi(b - a)$.

A segunda etapa consiste em verificar se esse valor sorteado de x será aceito ou rejeitado. Para isso, definimos uma função auxiliar constante do tipo $g(x) = c$, de modo que $g(x) \geq f(x)$, para qualquer valor de x no intervalo $[a : b]$. Sendo ξ_2 um outro número aleatório, aceitamos o valor amostrado para x se:

$$\xi_2 \leq \frac{f(x)}{g(x)} \quad (4.8)$$

Caso esta condição não seja satisfeita, um novo valor de x deve ser amostrado e o processo se repete.

A figura 4.7 ilustra a técnica da rejeição.

Embora a função $g(x)$ possa ser definida com qualquer valor constante, contanto que seja sempre maior do que $f(x)$, quanto mais próximas as duas funções forem, menor será a taxa de rejeição da técnica. O valor ideal para $g(x)$ é ser igual ao valor máximo assumido por $f(x)$ no intervalo em que é definida.

A eficiência E do método que emprega a técnica da rejeição é dada pela razão entre as áreas definidas sob as funções $f(x)$ e $g(x)$, tal que:

Figura 4.7: Técnica da Rejeição

$$E = \frac{\int_a^b f(x)dx}{(b-a)g(x)} = \frac{1}{(b-a)g(x)} \quad (4.9)$$

4.2.2 Gerador de números pseudoaleatórios

Um bom gerador de números pseudoaleatórios (do inglês Pseudo Random Number Generator, PRNG) é fundamental para um código que realize simulações através do Método de Monte Carlo.

Uma sequência de números pseudoaleatórios é composta por números que simulam a aleatoriedade, no entanto são gerados por um algoritmo e possuem uma sequência bem definida. Uma sequência de números pseudoaleatórios apresenta um bom resultado quando nela aplicado um teste de aleatoriedade, mas possui a vantagem de poder ser repetida quantas vezes for necessário. Já uma sequência de números realmente aleatórios, não pode ser repetida, pois não é gerada por um algoritmo.

A possibilidade de se prever o comportamento de uma sequência de números pseudoaleatórios e repeti-la é especialmente importante em simulações que usam o método

de Monte Carlo por uma questão de reprodutibilidade.

Em uma simulação de Monte Carlo que use números aleatórios, cada vez que a mesma simulação for realizada, um resultado diferente será obtido devido ao fato dos números sorteados não serem os mesmos. Obviamente, para um número de histórias suficientemente grande o valor estatístico tenderá a converter para o mesmo resultado, no entanto, cada história individualmente será diferente.

A reprodutibilidade das sequências de números pseudoaleatórios é importante quando se deseja realizar uma análise mais detalhada de uma simulação. Pois é possível saber exatamente a sequência de eventos que compõem cada história, ou então, ao se alterar algum parâmetro, saber o efeito que tal alteração trouxe na simulação, uma vez que a sequência de números deixa de ser uma variável do problema a ser analisada.

Um dos métodos de se gerar números pseudoaleatórios é através de Geradores Lineares Congruenciais (GLC). O método consiste em um algoritmo que recebe uma semente (número inteiro positivo) como dado de entrada e retorna um número real entre 0 e 1 e uma segunda semente. Assim, o sistema pode ser retroalimentado e gerar uma sequência de números entre 0 e 1 que serão os números pseudoaleatórios.

A estrutura de um GLC é representada pelo método recursivo,

$$S_n = (a.S_{n-1} + c)MODm, \quad (4.10)$$

onde a , c e m são parâmetros fixos, S é a semente e a função $(x)MODm$ representa o resto da divisão inteira de x por m .

Depois de obtida a semente, o número pseudoaleatório é definido por,

$$R = S/m \quad (4.11)$$

Vejamos um exemplo para o caso em que $a = 47$, $c = 1$, $S_0 = 1$, $m = 100$ na tabela 4.1:

Começando com a semente $S_0 = 1$, obtemos 48 como valor para a nova semente e 0.48 como primeiro número pseudoaleatório. Seguimos o processo até obtermos $S_{20} = S_0 = 1$. À partir do aparecimento dessa semente pela segunda vez, a mesma sequência de números gerados passa a se repetir. Dizemos então que esse gerador possui um período de 20 números pseudoaleatórios. O período de um gerador depende, não só do algoritmo nele usado, mas também dos parâmetros que o definem. Se tivéssemos no caso anterior,

para m , que seria $2^{32} - 1$ para sistemas computacionais de 32 bits e $2^{64} - 1$ para sistemas de 64 bits.

4.2.2.1 RANECU

Neste trabalho, o PRNG escolhido para ser usado nas simulações foi a adaptação para execução em GPU de um trabalho realizado por Badal em 2006 [7] que consistiu no desenvolvimento de uma versão paralela do RANECU com o intuito de permitir a execução em paralelo do código PENELOPE [7].

O gerador RANECU é usado em alguns softwares de Monte Carlo, dentre eles o PENELOPE, por possuir um período relativamente longo (2.3×10^{18} números) e boas propriedades estatísticas. Tais características tornam o RANECU uma boa opção de gerador para códigos que não necessitam de sequências muito longas de números pseudo-aleatórios. Desenvolvido por L'Ecuyer em 1988 [31], combina as sequências produzidas por dois geradores congruenciais lineares multiplicativos (MLCGs) para produzir uma terceira sequência.

Dadas as sequências $R_i^{(1)}$ e $R_i^{(2)}$, a terceira sequência é definida por:

$$R_i = (R_i^{(1)} - R_i^{(2)}) \text{MOD}(m^{(1)} - 1) \quad (4.12)$$

Cujo maior período possível é definido por $(m^{(1)} - 1)(m^{(2)} - 1)/2$. Para os parâmetros usados no RANECU (tabela 4.2), isso nos dá um período de $2305842648436451838 \approx 2.3 \times 10^{18}$ números.

Tabela 4.2: Parâmetros dos geradores MLCGs presentes no RANECU

	Módulo (m)	Multiplicador (a)
Gerador 1	2 147 483 563	40014
Gerador 2	2 147 483 399	40692

Existe também uma extensão do RANECU que trabalha com 3 geradores MLCGs para a qual o período é de aproximadamente 5×10^{27} números. Nela temos $m^{(3)} =$

2147482739 e $a^{(3)} = 45742$ e a sequência final é definida por:

$$R_i = (R_i^{(1)} - R_i^{(2)} + R_i^{(3)}) \text{MOD}(m^{(1)} - 1) \quad (4.13)$$

No entanto, por ser ligeiramente mais lenta, e as simulações de Monte Carlo tipicamente não precisarem de tantos números pseudoaleatórios, trabalharemos com a versão do código que possui apenas 2 geradores.

A figura 4.8 mostra o código em FORTRAN do gerador RANECU presente no código PENELOPE.

```

      FUNCTION RAND(DUMMY)
C   This is an adapted version of subroutine RANECU written by F. James
C   (Comput. Phys. Commun. 60 (1990) 329-344), which has been modified to
C   give a single random number at each call.
C
C   The 'seeds' ISEED1 and ISEED2 must be initialized in the main program
C   and transferred through the named common block /RSEED/.
C
C   Some compilers incorporate an intrinsic random number generator with
C   the same name (but with different argument lists). To avoid conflict,
C   it is advisable to declare RAND as an external function in all sub-
C   programs that call it.
C
      IMPLICIT DOUBLE PRECISION (A-H,O-Z), INTEGER*4 (I-N)
      PARAMETER (USCALE=1.0D0/2.147483563D9)
      COMMON/RSEED/ISEED1, ISEED2
C
      I1=ISEED1/53668
      ISEED1=40014*(ISEED1-I1*53668)-I1*12211
      IF(ISEED1.LT.0) ISEED1=ISEED1+2147483563
C
      I2=ISEED2/52774
      ISEED2=40692*(ISEED2-I2*52774)-I2*3791
      IF(ISEED2.LT.0) ISEED2=ISEED2+2147483399
C
      IZ=ISEED1-ISEED2
      IF (IZ.LT.1) IZ=IZ+2147483562
      RAND=IZ*USCALE
C
      RETURN
      END

```

Figura 4.8: Código em FORTRAN do gerador RANECU presente no código PENELOPE

4.2.2.2 Paralelização do gerador de números pseudoaleatórios

Os algoritmos para gerar números pseudoaleatórios baseados no método linear congruencial apresentam uma sequência de números com determinado período que passa a se repetir ciclicamente.

Existem dois grandes grupos principais de geradores de números pseudoaleatórios em paralelo (PPRNG) que são os geradores que trabalham com diferentes sequências de números, independentes entre si, e aqueles que trabalham com diferentes trechos de uma única sequência numérica. Os geradores do primeiro tipo podem trabalhar com o mesmo algoritmo, mas com parâmetros iniciais distintos.

Por sua vez, os geradores do segundo tipo podem ser divididos em dois subgrupos; os que tem um trecho da sequência destinada para cada gerador, e os que se utilizam da técnica conhecida como *leapfrogging* (saltos).

A técnica *leapfrogging* consiste em fazer com que cada gerador ‘salte’ o número de geradores existentes entre um número e outro, ou seja, se possuo M geradores, cada gerador R_i trabalhará com as sementes $[S_i, S_{i+M}, S_{i+2M}, \dots]$. Desse modo, não existe problema de sobreposição entre os geradores.

A outra técnica usada é a de segmentar a sequência original em diversas subsequências, deixando cada gerador responsável por um segmento. Se um PRNG possui uma sequência de $N.M$ números pseudoaleatórios, ela pode ser separada em M geradores com N números cada. Assim, cada gerador R_i trabalhará com as sementes $[S_{iN}, S_{iN+1}, S_{iN+2}, \dots, S_{iN+N-1}]$. Dessa forma, cada gerador poderá fazer uso de uma sequência única composta por N números. Naturalmente, caso esse valor seja excedido, o gerador ‘invade’ a sequência do gerador seguinte. Deve-se, portanto, tomar cuidado ao dimensionar a paralelização de um PRNG para que a quantidade de números disponíveis por gerador seja o suficiente para realizar a simulação desejada.

O PRNG presente neste trabalho é do tipo que trabalha com segmentos de uma sequência inicial.

Para que seja possível seccionar a sequência de números gerada por um gerador em M subsequências, é necessário termos os valores das enésimas sementes $[S_0, S_n, S_{2n}, S_{3n}, \dots, S_{M.n}]$ pois elas servirão como parâmetro inicial para cada gerador, que trabalhará com uma

sequência de n elementos.

Da eq. 4.10, quando temos $c=0$, obtemos que o valor de S_n é definido por:

$$S_n = (a^n \cdot S_0) \text{MOD} m, \quad (4.14)$$

Dessa forma, o valor de S_n pode ser obtido sem a necessidade de se calcular o valor das sementes intermediárias, o que faz com que o processo de paralelização do gerador RANECU não seja difícil. Na seção 5.1 o método de paralelização do gerador RANECU é descrito em detalhes.

4.2.3 Transporte de fótons

Na simulação do transporte de fótons, cada fóton é transportado individualmente. Sua história se inicia na fonte, ele realiza uma série de passos seguidos de interações, e tem sua história finalizada quando atinge uma energia mínima limite ou abandona a geometria do problema. Tipicamente, para energias iniciais da ordem de 1 MeV e meios com densidade próxima de $1g/cm^3$, um fóton sofre até 10 interações antes de ter sua história finalizada. Desse modo, a simulação da história de um fóton é realizada passo a passo, ou seja, todas as interações e passos são simulados. Diferentemente do caso de elétrons, que sofrem centenas de interações, e portanto, normalmente cada interação simulada corresponde a uma média dessas interações.

A simulação do transporte de um fóton é dividida nas seguintes etapas:

- Sorteio do passo ²
- Checagem das fronteiras
- Sorteio da interação
- Cálculo dos tallies ³

Essas etapas se repetem ciclicamente até que a história do fóton seja finalizada.

²O passo de uma partícula é o caminho que ela percorre entre as interações

³Os tallies são as informações macroscópicas calculadas a partir da simulação dos eventos microscópicos

4.2.3.1 Sorteio do passo

Ao se realizar o sorteio do passo de um fóton, deve-se sortear a direção de propagação, e o comprimento do passo. Usando coordenadas esféricas, isso significa sortear R, θ e φ em relação à posição atual da partícula, conforme ilustra a figura 4.9. Em seguida, esse valor é convertido para coordenadas cartesianas, seguindo a relação:

$$\Delta S = (\Delta x, \Delta y, \Delta z) = (R \sin \theta \cos \varphi, R \sin \theta \sin \varphi, R \cos \theta) \quad (4.15)$$

de modo que $\theta \in [0 : \pi]$ e $\varphi \in [0 : 2\pi]$.

Figura 4.9: As coordenadas θ e φ descrevem a direção de propagação da partícula

Para um sorteio isotrópico da direção de propagação das partículas, isto é, para que a projeção radial das partículas fique uniformemente distribuída sobre a superfície de uma esfera de raio R , a PDF utilizada para o sorteio da direção é dada por [32]:

$$f(\theta, \varphi) d\theta d\varphi = \frac{1}{4\pi} \sin \theta d\theta d\varphi = \left[\frac{\sin \theta}{2} d\theta \right] \left[\frac{1}{2\pi} d\varphi \right] \quad (4.16)$$

Isso nos mostra que θ e φ são variáveis independentes e podem ser amostradas pelo método da inversão. Desse modo, temos:

$$\theta = \cos^{-1}(1 - 2\xi_1) \rightarrow \cos(\theta) = 1 - 2\xi_1 \quad (4.17)$$

$$\varphi = 2\pi\xi_2 \quad (4.18)$$

A figura 4.10 mostra a distribuição de pontos sob uma esfera de raio 1 seguindo a PDF anterior à esquerda e, à direita, a mesma distribuição caso o sorteio fosse feito segundo as funções $\theta = \pi\xi_1$ e $\varphi = 2\pi\xi_2$.

Figura 4.10: Comparativo entre a distribuição dos pontos na superfície de uma esfera para o caso de uma distribuição isotrópica (à esquerda) e para um caso em que se amostra θ variando uniformemente entre θ e π (à direita)

Podemos perceber que, embora seguir o sorteio de $\theta = \pi\xi_1$ pareça intuitivo, a distribuição de pontos decorrente deste sorteio apresenta um acúmulo nas regiões dos polos.

Já para o sorteio de R , que descreve o comprimento do passo realizado pela partícula, a PDF que descreve o processo é $f(x) = \Sigma_T e^{-\Sigma_T x}$, onde Σ_T é a seção de choque macroscópica do material no qual a partícula se propaga. Pelo método da inversão, obtemos:

$$R = \frac{-\ln(1 - \xi)}{\Sigma_T} = \frac{-\ln(\xi')}{\Sigma_T} \quad (4.19)$$

A seção de choque macroscópica é uma grandeza que expressa o quanto um feixe de partículas é atenuado ao penetrar em um material. Para isso, ela relaciona as seções de choque microscópicas correspondentes a cada interação que a partícula pode sofrer com o material. A figura 4.11 mostra as seções de choque de fótons para a água [33].

Figura 4.11: Seção de choque para fótons na água, considerando-se o efeito fotoelétrico, produção de pares, espalhamento coerente e incoerente

4.2.3.2 Propagação da partícula

A partícula pode ser propagada através de duas formas distintas. A primeira delas é o método convencional, onde a partícula é deslocada pelas diversas células da

geometria, uma de cada vez. O segundo método é o ‘delta tracking’, mais conhecido como método de Woodcock, onde a partícula é propagada em um meio virtual homogêneo sem fronteiras.

Ambos os métodos foram implementados nesse trabalho e serão discutidos à seguir.

- Método convencional (método com paradas)

Pelo método convencional, após o passo da partícula ser sorteado, deve-se verificar se esse passo não implica em atravessar alguma das fronteiras que delimitam o meio no qual as partículas são propagadas. No caso da partícula cruzar alguma fronteira, ela é forçada a parar na interface e consultar as informações do novo material no qual ela está entrando.

Dessa forma, teremos três situações possíveis:

- **Partícula não cruzou nenhuma fronteira**

Caso seja verificado que a partícula realizou o passo sem interceptar nenhuma fronteira, inicia-se a próxima etapa da simulação que é o sorteio da interação que ela irá sofrer.

- **Partícula cruzou fronteira mas não mudou de material**

Caso seja verificado que, embora a partícula tenha interceptado alguma fronteira, o novo material continua sendo o mesmo (fronteira virtual), a partícula continua seu trajeto ignorando a existência dessa fronteira até que intercepte uma nova fronteira ou chegue à posição sorteada e, em seguida, sofra uma interação.

- **Partícula cruzou a fronteira e mudou de material**

Nesse caso, a partícula terá um novo passo sorteado mas, dessa vez, usando as informações de seção de choque do novo material. No entanto, a direção de propagação não deve ser alterada, sorteia-se apenas o novo comprimento R do passo.

- Método de Woodcock ou ‘delta tracking’

O Método de Woodcock [34] realiza a simulação do transporte da partícula ignorando as fronteiras existentes na geometria.

O caminho livre médio (λ) de uma partícula depende do material no qual ela se propaga. Tal característica está relacionada à seção de choque macroscópica do material (Σ_t), de modo que, quanto maior o valor de Σ_t menor o valor de λ . O método de Woodcock consiste em se considerar que a partícula é propagada em um meio homogêneo sem fronteiras composto pelo material do objeto simulador (meio) de maior Σ_t , ou seja, aquele em que a partícula apresenta o menor caminho livre médio. Caso esse livre caminho médio seja maior do que a distância média entre as fronteiras, existe um ganho de desempenho.

Para compensar a aproximação de meio homogêneo, é introduzida uma probabilidade de que a partícula sofra uma interação após realizar o passo (sem o método de Woodcock, considera-se 100% de chance de que haja a interação). A probabilidade de que ocorra uma interação é dada pela razão entre as seções de choque do meio (Σ_{Meio}) e a seção de choque máxima ($\Sigma_{Máx}$), ou seja;

$$Prob = \frac{\Sigma_{Meio}}{\Sigma_{Máx}} \quad (4.20)$$

Caso não haja interação, o passo é sorteado novamente até que a interação ocorra.

- Método de Woodcock Vs método convencional

Embora o método convencional obrigue a partícula a realizar uma parada toda vez que encontre uma fronteira, esse processo não é muito significativo (em termos de tempo computacional) para uma geometria composta por poucas fronteiras. Existe também a vantagem de sempre ocorrer uma interação após o passo ser concluído, e a existência de regiões com materiais de alta seção de choque não interfere na propagação da partícula em outras regiões da geometria.

O método de Woodcock, por sua vez, mostra-se eficiente para geometrias compostas por muitas fronteiras, principalmente quando essas fronteiras não representam uma fronteira física real do meio. Para meios heterogêneos, o método de Woodcock ainda mostra-se eficiente quando as seções de choque entre os materiais não são muito diferentes. No caso de seções de choque muito diferentes, além das partículas

serem propagadas com passos muito curtos, a probabilidade de interação também é baixa, fazendo que a eficiência do método caia.

Também existe a possibilidade de trabalhar com métodos mistos, como é o caso do código *Serpent* [10], que aplica uma combinação dos métodos de paradas e de *Woodcock* afim de obter melhor desempenho. Essa opção, no entanto, não está presente no código *CUBMC*.

4.2.3.3 Sorteio da interação

Para o transporte de fótons, quatro interações principais são simuladas (4.12). O Efeito Fotoelétrico, Produção de Pares, Espalhamento Rayleigh (coerente) e Espalhamento Compton (incoerente), sendo que a probabilidade de ocorrência de cada uma dessas interações é proporcional à sua seção de choque.

No decorrer das simulações de cada uma dessas interações, partículas secundárias como elétrons e pósitrons serão geradas. No entanto, o presente trabalho tem como proposta realizar apenas o transporte dos fótons. Desse modo, como aproximação inicial, sempre que uma partícula que não seja um fóton for gerada, sua energia será depositada localmente. Tal aproximação é justificada pelo fato do livre caminho médio do fóton ser ordens de grandeza maior do que o livre caminho médio do elétron. Dessa forma, mesmo que a simulação do transporte do elétron fosse realizada, haveria uma grande probabilidade de que o elétron acabasse depositando toda a sua energia antes de abandonar a célula (voxel) que foi gerado.

Ainda assim, tal aproximação apresentará desvios de modelos físicos mais completos, por não considerar os fótons que serão gerados por esses elétrons. Fótons esses que provavelmente abandonariam o voxel e depositariam sua energia em outro local.

- Efeito Fotoelétrico

No efeito fotoelétrico, o fóton interage com a eletrosfera do átomo sendo absorvido e transmitindo sua energia a um elétron que é ejetado. A conservação de momento se dá pelo recuo do átomo residual, e o elétron é ejetado com a energia do fóton (E) menos a energia de ligação que tinha com o átomo (U_i). Desse modo, para que seja possível a ejeção de elétrons mais fortemente ligados ao núcleo, é necessário que o

Figura 4.12: Representação esquemática das 4 interações simuladas pelo código CUBMC

fóton incidente possua maior energia.

Quando um elétron mais fortemente ligado ao núcleo é ejetado, outro elétron mais energético transicionará entre as camadas eletrônicas para ocupar a vacância gerada. Tal transição gera fótons de fluorescência (ou raio-x característico) cuja energia é igual a diferença entre as energias dos níveis atômicos envolvidos nessa transição. Caso esse fóton seja capturado antes de abandonar o átomo, ele pode ejetar um segundo elétron chamado de elétron Auger. Nota-se então que a fluorescência e a emissão de elétron Auger são processos que competem entre si. A figura 4.13 contém uma representação esquemática do efeito fotoelétrico.

A seção de choque fotoelétrica varia conforme a energia do fóton e o número atômico do átomo alvo. Ela é maior para fótons menos energéticos e átomos de maior número atômico. Conforme ilustra a figura 4.14.

Sendo um efeito predominante para fótons de baixa energia, a fluorescência decorrente do efeito fotoelétrico está, frequentemente, abaixo da energia de corte da simulação de Monte Carlo e não chega a ser simulada.

Uma vez que o fóton de fluorescência é de baixa energia e o modelo físico simpli-

Figura 4.13: Representação do efeito fotoelétrico: O fóton incidente (1) excitando o átomo e causando a ejeção do elétron (3) gerando uma vacância (2). Em seguida ocorre o relaxamento atômico seguido da emissão de fluorescência (4) que pode, por sua vez, ejetar um segundo elétron (5), chamado de elétron Auger

ficado usado nesse trabalho não engloba o transporte de elétrons, toda vez que a interação sorteada for o efeito fotoelétrico, toda a energia do fóton será depositada localmente e sua história será encerrada. Tal aproximação apresentará desvios significativos dependendo da energia do fóton. Os resultados obtidos serão apresentados e discutidos posteriormente.

- Produção de Pares

A produção de pares é um fenômeno no qual o fóton é absorvido pelo átomo alvo dando origem a um par elétron-pósitron. Para que isso seja possível, o fóton incidente necessita ter no mínimo a energia de repouso do elétron e pósitron somadas, $2m_e c^2$. A energia restante é dividida entre as energias cinéticas do elétron e o pósitron.

$$E_c = E_\gamma - 2m_e c^2 = E_c^+ + E_c^- \quad (4.21)$$

As equações que descrevem a seção de choque diferencial para a produção de pares

Figura 4.14: Seções de choque para o efeito fotoelétrico em função da energia para átomos de carbono, ferro e urânio

são baseadas em modelos semi-empíricos. Neste trabalho é usada a tabela de seção de choque gerada pelo código PENELOPE que, para a produção de pares, é obtida por tabelas da XCOM [33].

Uma vez que as simulações deste trabalho não englobam transporte de elétrons ou pósitrons, toda vez que a interação selecionada for a produção de pares, toda a energia do fóton será depositada localmente e sua história será encerrada. As consequências de tal aproximação serão discutidas posteriormente.

- Espalhamento Rayleigh (coerente)

O espalhamento coerente ocorre quando o fóton interage com a eletrosfera de um átomo mas não o excita ou ioniza. Ocorre a troca de momento entre o fóton e o átomo, mas ambos permanecem com a mesma energia. Tal espalhamento tem como análogo clássico uma colisão elástica entre bolas de bilhar.

Para fótons incidentes com energia muito maior do que a energia de ionização da camada K do átomo, a seção de choque diferencial para o espalhamento Rayleigh por unidade de ângulo sólido pode ser aproximada por:

$$\frac{d\sigma_{Ra}}{d\Omega} = \frac{r_e^2}{2}(1 + \cos^2 \theta)F^2(q, Z) \quad (4.22)$$

onde θ é o ângulo de espalhamento, r_e é o raio clássico do elétron e q é a magnitude da transferência de momento, definida por:

$$q = |p - p'| = 2(E/c) \sin(\theta/2) \quad (4.23)$$

onde p e p' são os momentos do fóton antes e após a interação e c é a velocidade da luz no vácuo.

A função $F(x, z)$ é o fator de forma atômico, que para átomos de densidade eletrônica esfericamente simétrica é dado por:

$$F(q, Z) = 4\pi \int_0^\infty \rho(r) \frac{\sin\left(\frac{qr}{\hbar}\right)}{\frac{qr}{\hbar}} r^2 dr \quad (4.24)$$

onde r é o raio, $\rho(r)$ é a densidade em função do raio e $\hbar$ é a Constante de Planck dividida por 2π . Descrevendo assim uma função monotonicamente decrescente em q , $F(q, Z)$ varia de $F(0, Z) = Z$ até $F(\infty, Z) = 0$.

Dessa forma, para fótons de baixa energia, a seção de choque para o espalhamento Rayleigh é dada por:

$$\sigma_{Ra} \approx \frac{8}{3}\pi r_e^2 Z^2 \quad (4.25)$$

E para fótons de alta energia:

$$\sigma_{Ra} \propto E^{-2} \quad (4.26)$$

Neste trabalho, o algoritmo usado para a simulação do espalhamento coerente trabalha com as seções de choque obtidas pelo código PENELOPE.

- Espalhamento Compton (incoerente)

O espalhamento Compton ocorre quando um fóton de energia E interage com um elétron que o absorve e emite um fóton secundário de energia E' . Em seguida, o elétron é ejetado com energia cinética dada por $E_e = E - E' - U_i$, onde U_i é a

energia de ligação do elétron. O análogo clássico do espalhamento Compton é o de uma colisão inelástica, onde o fóton transmite parte de sua energia para o elétron e é espalhado com um ângulo θ .

No caso do espalhamento ocorrer com um elétron livre em repouso, para que haja conservação de energia e momento, é possível relacionar a energia final do fóton (E') com o seu ângulo de espalhamento pela fórmula:

$$E' \equiv \frac{E}{1 + k(1 - \cos\theta)} \equiv E_c \quad (4.27)$$

onde $k = \frac{E}{m_e c^2}$.

A aproximação de Klein-Nishina [35] para a seção de choque diferencial do espalhamento Compton para elétrons livres em repouso é dada por:

$$\frac{d\sigma_{Co}^{KN}}{d\Omega} = \frac{r_e^2}{2} \left(\frac{E_c}{E} \right)^2 \left(\frac{E_c}{E} + \frac{E}{E_c} - \sin^2 \theta \right) \quad (4.28)$$

As tabelas de seção de choque usadas nesse trabalho baseiam-se no modelo de Klein-Nishina, mas incluem fatores de correção para serem aplicáveis a elétrons ligados e em movimento. Assim como todas as tabelas usadas neste trabalho, essas também são as mesmas utilizadas pelo código PENELOPE [5].

O espalhamento Compton é o principal responsável por deposição de energia para fótons com o intervalo de energia de interesse desse trabalho (de 1 KeV à 9 MeV). Desse modo, o algoritmo que simula a interação Compton usado foi extraído do código PENELOPE sem simplificações no que diz respeito ao tratamento dos fótons. Os elétrons decorrentes do efeito Compton são gerados mas, em seguida, tem sua energia depositada localmente. Essa aproximação é necessária pois o código CUBMC não transporta elétrons. As consequências de tal aproximação serão discutidas posteriormente.

Detalhes específicos da implementação do código serão apresentados na seção 5.

4.2.3.4 Cálculo dos tallies

Os tallies são os dados macroscópicos calculados através da simulação de eventos microscópicos da história de uma partícula, como fluxo de partículas, deposição de

energia, densidade de partículas, etc. Esse cálculo pode ser realizado para toda a geometria, ou apenas para algumas regiões de interesse.

Nessa etapa de simulação da história do fóton, as informações pertinentes são transferidas para contadores e passam a compor o grupo estatístico que fará parte dos resultados da simulação.

4.3 AMIGOBBrachy

O software AMIGOBBrachy [36] foi desenvolvido para criar uma poderosa interface de fácil utilização que integra os planos de tratamento clínicos com simulações de Monte Carlo, fornecendo os principais recursos necessários para processar e editar imagens, importar e editar planos de tratamento e definir parâmetros para simulações em Monte Carlo analisando seus resultados. O modelo de implementação atual, usa o código MCNP6 [4] para essas simulações, mas futuramente, também terá o CUBMC como uma opção de software simulador.

O AMIGOBBrachy foi desenvolvido usando a plataforma do MATLAB [37] versão 8.0 (Mathworks Inc., Natick, MA), e tem como objetivo fornecer ferramentas eficazes para processar imagens médicas, identificar as posições de parada, detectar agulhas, criar objetos simuladores usando o formato DICOM [38], criar arquivos de entrada para realizar simulações de MC com o código MCNP6 e analisar os resultados através de uma interface de fácil utilização, que é composta por 8 módulos principais com ferramentas interativas específicas. O software é compatível com o sistema de planejamento de tratamento (TPS) Nucletron OncentraTM [39] e Varian BrachyVisionTM (Varian Medical Systems, Palo Alto, CA), oferecendo a capacidade de importar planos de tratamento criados em ambos os TPS e realizar simulações utilizando o código MCNP6. As principais funcionalidades do software são:

- Importar ou criar uma sequência de imagens DICOM - atualmente imagens de tomografia computadorizada (TC) e ressonância magnética (RM) podem ser importadas, embora o usuário possa criar uma sequência de DICOM vazia e usar as ferramentas de desenho para criar objetos simuladores específicos, que podem ser exportados como imagens DICOM. Curvas automáticas de segmentação e de densidade de ca-

libração não estão disponíveis para imagens de ressonância magnética, que devem ser segmentados usando contornos ou limites definidos pelo usuário;

- Elaborar e/ou importar estruturas - contornos criados com um TPS comercial podem ser importados e modificados, por exemplo, para excluir ou incluir uma região específica, e, em seguida, combinar com a imagem. Novos contornos também podem ser definidos;
- Importar ou criar um plano de tratamento - tempos de parada e posições podem ser editados ou criados pelo usuário ao visualizar uma distribuição de dose TG-43U1 antes de realizar uma simulação.
- Criar um arquivo de entrada para o MCNP6 definindo os parâmetros da simulação, como, a energia de corte, física detalhada ou simples, tallies e regiões de interesse. Além disso, pode-se escolher entre as seguintes opções de transporte: transportar os fótons na água e calcular a dose em água ($D_{w,w}$), transportar os fótons no meio e calcular a dose em água ($D_{m,w}$) ou transportar os fótons no meio e calcular a dose no meio ($D_{m,m}$). A energia média em cada voxel pode ser calculada. Esta parte do AMIGOBrachy pode ser facilmente adotada para preparar os arquivos de entrada para outros códigos de Monte Carlo;
- Visualizar e comparar os resultados, que são importados automaticamente no final da simulação e podem ser analisados utilizando o AMIGOBrachy ou exportados usando o formato DICOM que pode ser lido por TPS comerciais.

Além das funcionalidades descritas, o AMIGOBrachy funciona como uma ferramenta auxiliar para criar arquivos de entrada à partir de geometrias do tipo mesh exportadas do AbaqusTM [40]. O código CUBMC vem sendo desenvolvido visando uma integração futura com o AMIGOBrachy, como uma das opções de software para simulação do transporte de radiação.

5 Resultados

5.1 Gerador de Números Pseudoaleatórios (PRNG)

O PRNG usado no código CUBMC é a versão adaptada para execução em paralelo do código RANECU. Tal procedimento foi explicado detalhadamente no artigo de Badal [7]. O algoritmo do gerador RANECU já foi apresentado na figura 4.8, pg.26, e a fórmula para se obter a n -ésima semente do gerador sem calcular as sementes intermediárias foi descrita pela eq. 4.14, pg.28. Sendo $S_i = (a^i S_0) \text{MOD} m$.

Tal procedimento pode ser aplicado aos geradores MLCGs presentes no RANECU, pois ambos trabalham com o parâmetro $c=0$. O programa seedsMLCG foi desenvolvido para gerar essas sementes espaçadas usadas no código RANECU. Em seu artigo [7], Badal apresenta uma tabela (tabela 5.1) demonstrativa dos valores das sementes espaçadas, começando-se com $S_1^{(1)} = S_1^{(2)} = 1$ e promovendo um espaçamento de 10^{15} entre as sementes para diferentes geradores. O algoritmo implementado no código CUBMC foi baseado no código seedsMLCG e apresentou os mesmos resultados presentes na tabela 5.1 para as sementes $S_i^{(1)}$ e $S_i^{(2)}$. As sementes $S_i^{(3)}$ não são calculadas no código CUBMC.

Após obter sementes concordantes com as da tabela 5.1, geradas pelo código seedsMLCG, o gerador de sementes presente no código CUBMC foi modificado para promover um espaçamento dinâmico entre as sementes, conforme a quantidade de geradores presentes em uma dada simulação. A sequência original de $\approx 2.3 \times 10^{18}$ números do gerador RANECU é dividida igualmente entre os geradores em execução no código. Desse modo, cada gerador trabalha com a maior sequência de números pseudoaleatórios possível, promovendo-se um aproveitamento máximo da sequência fornecida pelo gerador RANECU.

Tabela 5.1: Tabela demonstrativa do output do código seedsMLCG presente no artigo

RANECU Seed 1	RANECU Seed 2	Extension Seed
1	1	1
918 882 992	858 672 133	35 977 198
2 069 007 070	1 309 916 099	62 205 517
944 675 654	1 438 406 465	392 697 167
149 156 960	257 442 270	820 143 318
360 537 627	133 123 709	609 065 445
1 446 789 139	1 248 992 867	917 376 822
888 673 974	2 014 364 429	382 392 929
258 943	664 687 714	1 007 129 025
1 434 784 182	1 598 489 021	804 921 119
698 429 770	1 978 724 894	1 737 229 562

5.1.1 Comparativo de desempenho entre o RANECU e sua versão paralelizada

Após a implementação da versão paralelizada do código RANECU, foi estabelecido um comparativo do tempo de execução do código em sua versão linear, com a sua versão para execução paralela.

Para tal, a versão linear foi escrita em C, e é idêntica à versão presente no código PENELOPE mostrada na figura 4.8.

A versão modificada para a GPU passa por algumas etapas extras.

Inicialmente, as sementes para cada gerador são definidas por uma função executada fora da GPU, que é uma forma simplificada do programa seedsMLCG. Nessa função, foram definidas as sementes para 10^6 geradores afastadas entre si de 10^{12} .

Em seguida, essas sementes são transferidas para a GPU, onde os números aleatórios são gerados.

Embora a execução em paralelo na GPU seja mais rápida, existe um tempo

computacional dispendido para preparar as sementes e transferir a memória que não existe na execução linear. Isso faz com que a execução em paralelo seja mais lenta para se gerar poucos números pseudoaleatórios. No entanto, a medida em que se aumenta a quantidade de números gerados, esse tempo inicial passa a se tornar desprezível frente ao tempo total, e o ganho de desempenho presente na execução em paralelo na GPU passa a ser evidente.

Os números pseudoaleatórios foram gerados em um computador com Processador Intel Six-Core Core i7-980 de 3.33GHz e unidade de processamento gráfico NVIDIA GTX560Ti.

A tabela 5.3 mostra o comparativo dos tempos entre as versões linear e em paralelo do gerador RANECU, onde o termo ‘Ganho’ presente na tabela corresponde à razão entre esses tempos de execução.

Tabela 5.2: Comparativo dos tempos entre as versões linear e em paralelo do gerador RANECU

Números Gerados	t Linear (s)	t Paralelo (s)	Ganho
1E5	0.002	0.15	0.01x
1E6	0.02	0.14	0.13x
1E7	0.19	0.14	1.34x
1E8	1.88	0.16	11.45x
1E9	17.93	0.33	54.64x
1E10	178.02	1.85	96.12x
1E11	1729.17	17.10	101.09x
1E12	17258.32	169.40	101.88x

Podemos perceber que enquanto o método linear apresenta um tempo de execução diretamente proporcional à quantidade de números gerados, o método de execução paralela apresenta um tempo de latência de cerca de 0.15s. Tempo esse necessário para que as sementes sejam preparadas para os diversos geradores. Isso faz com que o método linear acabe sendo mais rápido para se gerar poucos números pseudoaleatórios.

Após esse tempo de latência, a versão de execução paralela do RANECU gera cerca de 5.9×10^9 números por segundo, enquanto sua versão linear gera 5.8×10^7 números

por segundo, fazendo com que o ganho de desempenho ao se gerar uma maior quantidade de números pseudoaleatórios seja de cerca de 100x.

5.1.2 Função *Rand*

Uma vez que o funcionamento adequado do gerador de sementes foi verificado, o gerador RANECU foi transferido para uma função do tipo `__device__` (que pode ser chamada durante a execução do *kernel*) chamada *Rand*. Essa função retorna um número pseudoaleatório entre 0 e 1 toda vez que for chamada. Os diferentes geradores são distribuídos entre os *threads* ativos na execução do código, de modo que cada *thread* gera uma sequência única de números.

A figura 5.1 mostra o algoritmo correspondente à função *Rand*.

```
__device__ void Rand(rnd r, int N, int idx);
#define _Rand Rand(r, N,idx);
__device__ void Rand(rnd r, int N, int idx)
{
    int I1, I2, IZ;

    if (idx<N){

        I1=r.Seed1[idx]/53668;
        r.Seed1[idx]=40014*(r.Seed1[idx]-I1*53668)-I1*12211;

        if(r.Seed1[idx]<0)r.Seed1[idx]=r.Seed1[idx]+2147483563;

        I2=r.Seed2[idx]/52774;
        r.Seed2[idx]=40692*(r.Seed2[idx]-I2*52774)-I2*3791;

        if(r.Seed2[idx]<0)r.Seed2[idx]=r.Seed2[idx]+2147483399;

        IZ=r.Seed1[idx]-r.Seed2[idx];

        if(IZ<1)IZ=IZ+2147483562;

        r.rand[idx]=IZ*Scale;

    }
}
```

Figura 5.1: Código em CUDA da função *Rand*

Na figura, podemos observar que o código é muito semelhante ao código em

FORTRAN do gerador RANECU fig.4.8, pg.26. As principais diferenças são os termos idx , N , e o fato de que as sementes $r.Seed1[idx]$ e $r.Seed2[idx]$ são vetores na versão do código escrita em CUDA.

O termo N designa o número de *threads* que serão executados no código. Esse número não significa necessariamente o número de *threads* ativos, mas sim o grau de paralelização da simulação. Para N *threads* teremos N geradores diferentes.

O termo idx representa o *thread* que está sendo executado no momento, variando de 0 a N .

As sementes, por sua vez, são vetores pois cada *thread* acessará o valor de semente correspondente ao seu gerador. Mesmo que todos os geradores sejam executados simultaneamente, cada um deles acessará o valor de semente correspondente ao seu *thread* de índice idx .

As variáveis $I1$, $I2$ e IZ são definidas no escopo de execução da função *Rand*, podendo ter seus valores apagados após a função ser executada. Isso permite que elas sejam alocadas na memória de registro dos *threads*, fazendo com que seu acesso seja mais rápido. No entanto, os valores das sementes devem ser preservados entre as chamadas da função *Rand* para que elas retornem o próximo número da sequência gerado pelo *thread*. Isso faz com que as sementes devam ser alocadas na memória global da GPU, de acesso mais lento.

5.2 Cálculo de π

Tendo definido a função *Rand*, o passo seguinte é implementá-la em algum código simples para verificar se esta se comporta corretamente. Para isso, foi escolhido um algoritmo que calcula o valor de π pelo Método de Monte Carlo.

Suponha uma circunferência de raio $R = 1$ inscrita em um quadrado de lado $L = 2R = 2$. A área da circunferência é dada por:

$$S_o = \pi r^2 \tag{5.1}$$

e a área do quadrado é dada por:

$$S_q = (2r)^2 = 4r^2 \quad (5.2)$$

Desse modo, a razão entre a área do círculo e a área do quadrado é $\pi/4$.

Nós não conseguimos determinar a área da circunferência de maneira exata sem saber o valor de π , mas se sortearmos um ponto aleatoriamente dentro do quadrado, sabemos que a probabilidade dele cair na área compreendida pela circunferência é proporcional à sua área. Dessa forma, se de um total de N pontos sorteados aleatoriamente, N_o caírem na região compreendida pela circunferência, temos que:

$$\frac{N_o}{N} \approx \pi/4 \quad (5.3)$$

Assim, o valor de π pode ser definido em função da razão entre os pontos que caem na área do círculo e o número total de pontos sorteados.

Para que o algoritmo fique mais simples e, portanto, seja executado mais rapidamente, os pontos são sorteados em $1/4$ da área total, apenas na região do primeiro quadrante da circunferência, conforme ilustra a figura 5.2.

Figura 5.2: Representação gráfica de pontos distribuídos aleatoriamente para a determinação de π

Como $R = 1$, o sorteio de um ponto aleatório no quadrado resume-se a $x = \xi_1$, e $y = \xi_2$. Em seguida, temos dois contadores, um deles (N) que é acionado toda vez que um ponto é sorteado, e um segundo (N_o) que é acionado apenas se $x^2 + y^2 < 1$, ou seja, quando o ponto sorteado está na área do círculo.

Finalmente, o valor de π é estimado por:

$$\pi \approx 4 \frac{N_o}{N} \quad (5.4)$$

Embora seja um problema bem simples, o cálculo de π pelo Método de Monte Carlo já incorpora os principais elementos presentes em simulações mais complexas. Além disso, como o valor de π é bem definido, é possível saber se a simulação está funcionando corretamente e avaliar a convergência de seu valor em função do número de pontos sorteados, mesmo após os números pseudoaleatórios começarem a se repetir. Também é possível fazer comparativos de análise de desempenho entre CPU e GPU.

Em um primeiro teste, usamos 10^5 geradores em paralelo espaçados de 10^{12} sementes entre si, sorteando um total de 10^9 pontos. Dessa forma, cada gerador consumiu 2×10^4 números pseudoaleatórios (um para o sorteio de X e outro para Y), estando longe da zona em que um gerador começa a invadir a sequência do outro. A tabela 5.3 apresenta a convergência para o valor de π em função da quantidade de pontos sorteados.

Podemos notar que, embora apresente uma pequena oscilação, o valor da aproximação de π tende a convergir para o valor de π conforme o número de pontos sorteados aumenta. A figura 5.3 apresenta a convergência para o valor de π para os primeiros 1000 pontos sorteados em uma simulação contendo apenas um gerador pseudoaleatório.

Em seguida, o mesmo teste de convergência foi feito para geradores com diferentes espaçamentos entre si.

Foram usados 10^5 geradores para sortear 2×10^{10} pontos, de modo que cada gerador ficou responsável por gerar 2×10^5 pontos, consumindo assim, 4×10^5 números pseudoaleatórios. O espaçamento entre os geradores foi dado por 10^K , onde $k = 0, 2, 4, 6, 8, 10$ e 12 . Dessa forma, como cada gerador usará 4×10^5 números, para os valores de $k = 0, 2$ e 4 , cada gerador começará a usar a sequência do gerador adjacente e é esperado que haja problemas de convergência.

Tabela 5.3: Convergência do valor de π

Número de Pontos	Aproximação de π	Desvio
10^1	2.7999	-10.876 %
10^2	3.2799	4.402 %
10^3	3.1760	1.095 %
10^4	3.1456	0.128 %
10^5	3.1368	-0.153 %
10^6	3.1397	-0.060 %
10^7	3.1408	-0.025 %
10^8	3.1412	-0.012 %
10^9	3.1415	-0.003 %

Figura 5.3: Convergência dos primeiros 1000 pontos para o valor de π em uma simulação com apenas um gerador pseudoaleatório

A figura 5.4 mostra claros problemas de convergência para $k = 0$ e $k = 2$, conforme o esperado.

A figura 5.5 apresenta uma vista aproximada das mesmas simulações, onde

Figura 5.4: Simulação de Monte Carlo para estimativa do valor de π usando diferentes configurações de geradores de números pseudoaleatórios.

podemos observar que a simulação com $k = 4$ convergiu para um valor diferente de π . Enquanto as simulações com $k > 5$ (sem conflito entre os geradores) apresentam uma convergência adequada.

Figura 5.5: Detalhe da simulação de Monte Carlo para estimativa do valor de Pi usando diferentes configurações de geradores de números pseudoaleatórios.

Conforme o esperado, as simulações com geradores de sementes iniciais pouco afastadas entre si apresentaram problemas de convergência ao se estimar o valor de π . Dessa forma, podemos concluir que a distribuição das sementes entre os geradores apresenta uma grande influência no resultado das simulações. Por esse motivo, o código CUBMC adota um modelo dinâmico de espaçamento entre os geradores, a sequência original do RANECU é dividida igualmente entre o número de geradores presentes em uma dada simulação, garantindo assim, que cada gerador tenha a maior sequência de números possível antes de invadir a sequência do gerador adjacente.

As diferenças nos tempos de execução das simulações em CPU e GPU seguiram o mesmo comportamento observado pelo gerador de números pseudoaleatórios. Para simulações com maior número de pontos sorteados (acima de 10^9) o ganho de desempenho da GPU foi próximo a 100x.

5.3 CUBMC

5.3.1 Aspectos Gerais

O código CUBMC foi desenvolvido com o intuito de utilizar a capacidade de processamento paralelo das GPUs para atingir o maior ganho de desempenho possível em simulações de transporte de fótons através do Método de Monte Carlo. Para isso, o transporte é realizado inicialmente em geometrias de baixa complexidade, seguindo modelos físicos simplificados. Uma vez que o código apresente resultados satisfatórios dentro de suas limitações, seu grau de complexidade será aumentado a fim de obtermos um sistema mais compreensivo e versátil.

Visando otimizações futuras, o código apresenta estrutura modular, permitindo que cada módulo possa ser substituído por um modelo mais complexo e abrangente em versões futuras.

5.3.2 Geometria

Desde seu primeiro modelo, o código CUBMC foi desenvolvido para simular o transporte de fótons em uma geometria descrita por voxels (voxel é a extrapolação tridimensional do pixel. Do mesmo modo que um conjunto de quadrados (pixels) podem formar uma imagem bidimensional, um conjunto de paralelepípedos (voxels) definem uma geometria tridimensional). Esse modo de descrever a geometria foi escolhido devido à sua simplicidade, mas que, no entanto, possibilita descrever facilmente geometrias complexas. A opção por voxels também facilitará a implementação de objetos simuladores provenientes de modelos anatômicos obtidos por imagens de tomografia computadorizada (CT), possibilitando assim, uma comunicação entre o código CUBMC e o sistema de planejamento AMIGOBachy descrito na seção 4.3.

No modelo atual do código CUBMC, os voxels são descritos por paralelepípedos de mesmo tamanho que podem ter dimensões distintas em X,Y e Z. O motivo dessa restrição está relacionado à rotina de checagem das fronteiras, e será abordado na seção 5.3.6. No momento, existe essa restrição, mas será removida futuramente.

Cada voxel da geometria é relacionado a um elemento inteiro de um vetor linear MAT que designa o material que o constitui. Dessa forma, é possível se determinar o material de cada voxel individualmente podendo, inclusive, cada voxel ser de um material diferente, dentro da limitação de memória da GPU para armazenar as tabelas de seção de choque desses materiais.

O índice de cada voxel no vetor linear é dado por:

$$ID = i + j * DimX + k * DimX * DimY; \quad (5.5)$$

onde i, j, k são, respectivamente os índices x, y e z do voxel, e DimX e DimY o número de voxels presentes em X e Y descrevendo a geometria.

Como os índices dos voxels são iniciados em zero, em uma geometria contendo N voxels, estes serão identificados por ID variando de 0 a N-1, o que, tipicamente, implica em um vetor MAT contendo algo entre 10^6 e 10^9 elementos dependendo do problema.

Apenas a memória global e a memória de textura são capazes de armazenar vetores com esse número de elementos (Os diferentes tipos de memória encontram-se

descritos na seção 4.1.2.1, pg 13). Atualmente, o vetor MAT encontra-se alocado na memória global, mas será movido para a memória de textura em futuras versões do código. Essa alteração deve resultar em uma leitura mais rápida do vetor, uma vez que a memória de textura possui cache de leitura dedicado.

A versão inicial do vetor MAT contém apenas elementos com um valor inteiro que serve como índice para referenciar o material que constitui o voxel. Dessa forma, voxels contendo materiais de mesma composição atômica mas com diferentes densidades são referenciados como sendo materiais diferentes, e consultam tabelas de seção de choque distintas.

A segunda versão do vetor MAT (que ainda será implementada) compreende materiais para os quais os valores das seções de choque macroscópicas variam linearmente com sua densidade, ou seja, para materiais de mesma composição atômica, o valor de suas seções de choque é diretamente proporcional à sua densidade. Assim, o vetor MAT será composto por elementos reais contendo, no mesmo número, informações sobre o índice do material e sua densidade. Dessa forma, materiais de mesma composição atômica consultarão a mesma tabela e terão, em seguida, os valores corrigidos segundo a sua densidade.

Para um material de índice X e densidade ρ , o valor contido no vetor MAT será do tipo:

$$MAT[ID] = 100X + \rho \quad (5.6)$$

Desse modo, os valores do índice do material e sua densidade podem ser obtidos pelas expressões:

$$X = \text{int}(MAT[ID]/100) \quad (5.7)$$

$$\rho = MAT[ID] - 100X \quad (5.8)$$

onde $\text{int}()$ é a função que retorna a parte inteira de um número racional.

Naturalmente, é possível usar dois vetores distintos para armazenar as informações de material e densidade. No entanto, isso implicaria em um maior uso de memória da GPU, uma vez que seria necessário alocar um vetor de números inteiros (materiais) e outro de números reais (densidades), e também implicaria em duas leituras

em vez de apenas uma. A GPU é significativamente mais lenta para realizar leituras do que operações, de modo que separar as informações de material e densidade de um único número é um processo mais rápido do que consultar duas informações alocadas em vetores distintos.

Independentemente do formato do vetor MAT, ele não passa de uma forma de armazenar os índices referentes às tabelas de seção de choque a serem consultadas durante a execução do código.

5.3.3 Tabelas de Seção de choque

As tabelas de seção de choque usadas no código CUBMC são geradas pelo código PENELOPE [5]. Entretanto, as tabelas geradas possuem mais dados do que são necessários para o código CUBMC, pois ele se restringe ao transporte de fótons, e apresenta diversas simplificações. Desse modo, da tabela originalmente gerada, o código carrega apenas o campo referente aos coeficientes de atenuação de massa para as interações simuladas (Espalhamento Coerente e Incoerente, Efeito Fotoelétrico e Produção de Pares) e alguns parâmetros a serem usados durante a rotina dessas interações. Esses parâmetros serão discutidos em detalhes quando o código que simula cada interação for abordado na seção 5.3.7.

Os coeficientes de atenuação de massa (μ) tem unidade de cm^2/g e representam a intensidade com que um fóton de uma determinada energia é absorvido por um material em decorrência de alguma interação. A tabela 5.4 apresenta o início da tabela com os coeficientes de atenuação de massa gerada para a água, pelo código PENELOPE e a figura 5.6 apresenta o gráfico da tabela completa.

Na tabela 5.4, temos os coeficientes gerados para cada interação, e o coeficiente total que é a soma deles. Dessa forma, o coeficiente de atenuação de massa total representa a probabilidade de que o fóton sofra uma interação qualquer, e os coeficientes parciais representam a probabilidade de cada interação.

O coeficiente de atenuação de massa total está relacionado ao livre caminho médio da partícula e seu uso é descrito na seção 4.2.3.1. Já o uso dos coeficientes parciais está relacionado ao sorteio de qual tipo de interação a partícula irá sofrer, e é descrito na

Tabela 5.4: Coeficientes de atenuação de massa gerada para a água, pelo código PEN-LOPE, para energias entre 50eV e 100eV. Onde $1mtu = 1g/cm^2$

Energy (eV)	Rayleigh (1/mtu)	Compton (1/mtu)	Photoelect. (1/mtu)	Pair (1/mtu)	Total (1/mtu)
5.00000E+01	8.17311E-01	3.34289E-13	2.12557E+05	1.00000E-35	2.12558E+05
5.23318E+01	8.42182E-01	3.34289E-13	1.99032E+05	1.00000E-35	1.99033E+05
5.47723E+01	8.67704E-01	3.34289E-13	1.85839E+05	1.00000E-35	1.85840E+05
5.73266E+01	8.89982E-01	3.34288E-13	1.72970E+05	1.00000E-35	1.72971E+05
6.00000E+01	9.10388E-01	3.34289E-13	1.60538E+05	1.00000E-35	1.60539E+05
6.23574E+01	9.26446E-01	3.34292E-13	1.50367E+05	1.00000E-35	1.50368E+05
6.48074E+01	9.40553E-01	3.34294E-13	1.40557E+05	1.00000E-35	1.40558E+05
7.00000E+01	9.65099E-01	3.34288E-13	1.22186E+05	1.00000E-35	1.22187E+05
7.48331E+01	9.82081E-01	3.34273E-13	1.07639E+05	1.00000E-35	1.07640E+05
8.00000E+01	9.96061E-01	3.34303E-13	9.44006E+04	1.00000E-35	9.44016E+04
8.48528E+01	1.00454E+00	3.34316E-13	8.38364E+04	1.00000E-35	8.38374E+04
9.00000E+01	1.01090E+00	3.34279E-13	7.42696E+04	1.00000E-35	7.42706E+04
1.00000E+02	1.01747E+00	3.34433E-13	5.94737E+04	1.00000E-35	5.94747E+04

seção 4.2.3.3.

Ao se executar o código CUBMC, os dados contidos nessas tabelas são transferidos para a memória constante da GPU, permitindo assim, um rápido acesso pelo *thread*. No entanto, a quantidade de memória constante presente na GPU é limitada a 64KB, de forma que, atualmente, o código CUBMC é capaz de simular uma geometria composta por até três materiais. Essa quantidade de materiais não é suficiente para simulações que tenham como objeto simulador um modelo anatômico proveniente de uma imagem de tomografia computadorizada. Dessa forma, em uma versão futura do código, será necessário armazenar as tabelas de materiais na memória de textura da GPU.

Figura 5.6: Coeficientes de atenuação de massa para a água gerados pelo código PENELOPE

5.3.4 Modelos de fonte de radiação

O módulo da fonte define onde os fótons são originados e qual a sua energia inicial, permite múltiplas fontes com formatos tridimensionais diferentes entre si. Também aceita que as energias sejam descritas por espectros, representando assim fontes reais.

A descrição da fonte é realizada por um *kernel* à parte do *kernel* principal que realiza o transporte da radiação e as informações são transferidas por memória global. Dessa forma, antes que o *kernel* de transporte seja iniciado, a posição e energia inicial de cada partícula a ser transportada já foi definida pelo *kernel* da fonte.

A fonte pode ser definida com qualquer forma desejada, pois não tem sua geometria vinculada à geometria do problema, podendo, inclusive, estar localizada pon-

tualmente na fronteira entre voxels.

Distribuições de probabilidade espaciais, ou referentes ao espectro de energia também podem ser adotadas, no entanto, o modelo atual apresenta duas restrições.

A primeira restrição é referente à localização espacial. O *kernel* de transporte finaliza a história de uma partícula quando ela se encontra fora da geometria descrita pelos voxels. Consequentemente, não é possível definir uma fonte externa à geometria do objeto simulador, pois a partícula já iniciaria sua história estando fora da geometria. Para se simular uma fonte externa, é necessário preencher o material entre a fonte e a região de interesse com vácuo. Tal processo é custoso, pois além de ocupar memória com informação de voxels que não são de interesse, as partículas serão transportadas nessa região.

Esse processo é especialmente custoso devido ao fato de que todos os voxels da geometria são definidos por paralelepípedos de mesmo tamanho. O motivo dos voxels serem definidos assim será abordado na seção 5.3.6, mas o fato dos voxels serem todos do mesmo tamanho implica que a partícula simulada leva quase o mesmo tempo computacional para ser transportada em um meio homogêneo de vácuo, quanto em uma geometria complexa. Essa forma de definir os voxels será alterada em versões futuras do código CUBMC mas, no momento, existe essa restrição.

A segunda restrição referente à modelagem da fonte é que, por estar dissociada da geometria do objeto simulador, ela não inclui informações de material. Caso queira se definir algum material para a fonte, ele deve ser descrito juntamente aos outros materiais que compõem o objeto simulador e, portanto, ser definido por voxels. Embora a geometria da fonte possa ser descrita por qualquer função desejada, o material encontra-se vinculado ao modelo de voxels.

5.3.5 Sorteio do passo

Após o *kernel* de transporte ser iniciado, a simulação da partícula passará ciclicamente por uma série de processos que descreverão sua história, até que ela abandone a geometria ou atinja a energia de corte. O sorteio do passo é o primeiro destes processos.

O passo de uma partícula é o deslocamento que ela sofrerá entre duas interações

consecutivas e é realizado em coordenadas esféricas. Para isso, três valores devem ser definidos, R (Raio), θ (ângulo polar) e φ (ângulo azimutal). O sorteio de θ e φ é realizado segundo uma distribuição de probabilidade, que pode ser isotrópica, ou dependente da interação ocorrida, já o sorteio do Raio é realizado levando-se em consideração o livre caminho médio da partícula no meio e, para tal, depende do material no qual o fóton se propaga.

A forma de se amostrar essas variáveis já foi demonstrada na seção 4.2.3 pelas equações 4.17, 4.18 e 4.19.

Para que o Raio seja sorteado, é necessário consultar a tabela de materiais para se obter o valor da seção de choque macroscópica para fótons daquela energia, que é o coeficiente de atenuação de massa total multiplicado pela densidade do material. Embora a tabela gerada pelo PENELOPE forneça os coeficientes de atenuação de massa, ao ser transferida para a GPU, seu valor é multiplicado pela densidade do material para que o coeficiente de atenuação de massa total corresponda à seção de choque macroscópica.

A figura 5.7 mostra o trecho do código que realiza a leitura da tabela e, em seguida, sorteia o valor do Raio.

```
ID= floor(X)+floor(Y)*DimX+floor(Z)*DimX*DimY;
for(i=DivMat[d.MAT[ID]]-1; i>=0;i--){
    if(Energia[d.MAT[ID]][i]<E){
        KE=i+1;
        break;
    }
}

P1=SGRA[d.MAT[ID]][KE];
P2=SGC0[d.MAT[ID]][KE];
P3=SGPH[d.MAT[ID]][KE];
P4=SGPP[d.MAT[ID]][KE];
ST=P1+P2+P3+P4;
_Rand;
Raio=-1e0f*log(r.rand[idx])/ST;
```

Figura 5.7: Trecho do código CUBMC que consulta a tabela de materiais e realiza o sorteio do comprimento do passo dado pelo fóton

A primeira linha do código presente na figura 5.7 calcula o índice ID do voxel no qual a partícula se encontra. Em seguida, o loop ‘for’ realiza a leitura da tabela do

material correspondente ao voxel, até obter qual é a linha (KE) da tabela, que é o índice do vetor no qual encontraremos as informações da tabela de seção de choque correspondentes à energia do fóton.

Após o valor de KE ser determinado, os valores $P1$, $P2$, $P3$ e $P4$ correspondentes, respectivamente, às seções de choque parciais para o Espalhamento Rayleigh, Espalhamento Compton, Efeito Fotoelétrico e Produção de Pares são determinados, e a Seção de choque macroscópica (ST) é dada como a soma desses termos. Embora a tabela gerada pelo código PENELOPE já forneça o valor da seção de choque total, é preferível economizar memória da GPU realizando a soma dos termos parciais.

No caso da simulação ser realizada segundo o Método de Woodcock, é calculado o valor de ST para cada um dos materiais presentes na geometria e adotado o maior valor obtido.

Após obter o valor de ST , a função RAND é executada para gerar o número pseudoaleatório $r.rand[idx]$, e o valor do Raio é calculado conforme a fórmula 4.19.

Em seguida, é realizado o sorteio de $\cos \theta$ e φ e a posição final (X_f , Y_f , Z_f) da partícula é determinada, conforme o código mostrado na figura 5.8.

```
_Rand;
_Fi=r.rand[idx]*PI2;

_Rand;
_Cos_Teta=1E0f-2E0f*r.rand[idx];

Xf=X+(Raio/Vx)*sqrt(1-(Cos_Teta*Cos_Teta))*cos(Fi);
Yf=Y+(Raio/Vy)*sqrt(1-(Cos_Teta*Cos_Teta))*sin(Fi);
Zf=Z+(Raio/Vz)*Cos_Teta;
```

Figura 5.8: Trecho do código CUBMC que sorteia a posição (X_f , Y_f , Z_f) da partícula após o passo

Da figura 5.8 podemos observar dois detalhes significantes deste trecho de código. O primeiro deles é que os ângulos θ e φ são definidos em relação a um eixo de coordenadas fixo. Como o sorteio da direção de propagação das partículas é feito na maioria dos casos de forma isotrópica, isso não é um problema. No entanto, após a partícula sofrer uma interação, é possível que exista uma nova distribuição de probabilidade para o

novo sentido de deslocamento, e essa distribuição trabalha com a referência no sentido de propagação que ela tinha antes de sofrer a interação. Nesses casos, é necessário realizar uma rotação no sistema de coordenadas.

O segundo detalhe presente no trecho de código mostrado na figura 5.8 é o fato de existir um fator de escala para o Raio. Podemos observar que o valor do Raio é dividido por Vx , Vy e Vz , que representam as dimensões em centímetros dos voxels em X , Y e Z , respectivamente. Esse fator de escala é necessário pois, durante a execução do *kernel* de transporte, os voxels são considerados sempre como sendo cubos de lado igual a $1cm$. Isso faz com que a checagem de fronteiras seja mais simples, pois se o valor inteiro da posição da partícula mudou, significa que ela se encontra em outro voxel. No entanto, é necessário corrigir o valor do passo para que o deslocamento dos fótons no espaço dos voxels corresponda ao deslocamento no espaço real. Devido à essa aproximação usada para facilitar a checagem de fronteiras, todos os voxels devem ter as mesmas dimensões. Para que os voxels fossem de tamanhos diferentes, Vx , Vy e Vz deveriam ser substituídos por vetores, correspondendo a diferentes fatores de escala para cada voxel. Essa alteração será implementada em futuras versões do código CUBMC.

5.3.6 Fronteiras

Após realizar o passo, o segundo módulo do *kernel* de transporte verifica se a partícula não cruzou alguma fronteira. Caso a simulação esteja sendo realizada segundo o método de Woodcock, o módulo de checagem de fronteiras não é executado.

Devido ao fator de escala usado que faz com que todos os voxels sejam tratados como cubos de lado $1cm$, a condição que define se a partícula cruzou a fronteira de algum voxel é bastante simples. Basta verificar em X , Y e Z se a componente inteira da posição final e inicial é a mesma. Caso ela tenha mudado, significa que a partícula trocou de voxel.

A figura 5.9 mostra o trecho de código que checa se houve colisão. A função $floor(X)$ retorna apenas o valor inteiro de um número X . No entanto, não existe arredondamento, por exemplo, $floor(1.999999)$ é 1, dessa forma, ao compararmos $floor(X)$ e $floor(Xf)$ não existe fatores de arredondamento por X ou Xf estarem próximos à borda

do voxel.

```
Colisao=0;
if(floor(Xf)!=floor(X) || floor(Yf)!=floor(Y) || floor(Zf)!=floor(Z)){

    #ifdef _PRINT
    //    printf("CRUZOU FRONTEIRA \n");
    #endif

    Xp=Xf;
    Yp=Yf;
    Zp=Zf;
    Colisao=1;
}
```

Figura 5.9: Trecho do código CUBMC que checa se a partícula cruzou a fronteira do voxel ao realizar o passo

Caso seja verificado que a partícula cruzou alguma fronteira ao realizar o passo, inicia-se uma rotina que calcula a posição X_p , Y_p e Z_p , que correspondem às coordenadas em que a fronteira foi interceptada.

A figura 5.10 representa o trecho do código que realiza a projeção de X_p , Y_p e Z_p , para o caso da partícula estar situada em um voxel de índice X diferente.

```
if(floor(Xp)<floor(X)){
    Crr=abs((X-floor(X))/(Xp-X));
    Xp=X+Crr*(Xp-X);
    Yp=Y+Crr*(Yp-Y);
    Zp=Z+Crr*(Zp-Z);
}
if(floor(Xp)>floor(X)){
    Crr=abs((ceil(X)-X)/(Xp-X));
    Xp=X+Crr*(Xp-X);
    Yp=Y+Crr*(Yp-Y);
    Zp=Z+Crr*(Zp-Z);
}
```

Figura 5.10: Trecho do código CUBMC que realiza a projeção de X_p , Y_p e Z_p para uma partícula que se encontre em um voxel de índice X diferente do voxel de origem

O processo de projeção realizado pelo código presente na figura 5.10 checa se a partícula avançou para um voxel de índice X maior ou menor, e corrige os valores para X_p , Y_p e Z_p fazendo com que eles correspondam ao ponto de interceptação do voxel de índice X adjacente. Em seguida, o processo é realizado caso o voxel também possua índice

diferente em Y e em Z . Após o processo ser finalizado, os pontos X_p , Y_p e Z_p correspondem ao ponto (na interface) em que a partícula cruza a fronteira do voxel inicial.

A figura 5.11 representa esse processo de uma forma gráfica. Inicialmente, a partícula que estava na posição 1 tem seu passo sorteado para a posição 2. Após o primeiro ajuste ela tem sua posição projetada em 3, e em seguida, em 4, na fronteira do voxel.

Figura 5.11: Representação gráfica bidimensional do processo de projeção para uma partícula que cruzou a fronteira entre voxels. A partícula que se encontrava na posição 1 teve seu passo sorteado para a posição 2. Em seguida, é projetada em 3 e em 4, sendo obrigada a parar na fronteira do voxel de origem.

Feita a projeção na fronteira do voxel, o material entre os voxels adjacentes é consultado. Caso os voxels sejam do mesmo material, a partícula passa da posição 4 para a 3 e o material entre os voxels adjacentes é consultado novamente. Esse processo se repete

até que a partícula atinja a posição final, ou seja detectado que ela está entrando em um meio com outro material. Nesse caso, um novo valor de Raio deve ser sorteado usando-se para o cálculo a seção de choque total do material no qual ela passa a se propagar. Apenas o valor do Raio é sorteado novamente, a partícula continua se propagando na mesma direção em que já vinha se propagando.

Esse processo de checagem de fronteiras e novos sorteios de Raio é realizado até que a partícula atinja a posição sorteada. Nesse ponto deverá ocorrer uma interação.

5.3.7 Interações

Após ser definido que a partícula sofrerá uma interação, é necessário sortear qual será a interação sofrida. Para isso, consultamos novamente os valores dos coeficientes presentes na tabela de seção de choque. Dessa vez, não será necessário ler as tabelas, pois os valores dos coeficientes já haviam sido lidos para se obter a seção de choque macroscópica, e ainda se encontram armazenados na memória de registro do *thread*. Conforme podemos observar no trecho de código presente na figura 5.12, ainda temos armazenados na memória o valor das variáveis *ST*, *P1*, *P2*, *P3* e *P4*, que correspondem, respectivamente, à seção de choque macroscópica, e às seções de choque parciais para o Espalhamento Rayleigh, Espalhamento Compton, Efeito Fotoelétrico e Produção de Pares.

```
if(Interacao==1){
ID= floor(X)+floor(Y)*DimX+floor(Z)*DimX*DimY;
_Rand;
STS=ST*r.rand[idx];

if(STS<=P1) lalala=1;
if(STS>P1 && STS<=P1+P2) lalala=2;
if(STS>P1+P2 && STS<=P1+P2+P3) lalala=3;
if(STS>P1+P2+P3) lalala=4;
```

Figura 5.12: Trecho do código CUBMC que realiza o sorteio entre as interações simuladas

Para realizar a amostragem das interações, é utilizado o método discreto (descrito na seção 4.2.1.2), onde o valor *STS* é uma variável pseudoaleatória uniformemente distribuída entre 0 e *ST*. Desse modo, a probabilidade de sorteio de cada uma das interações é proporcional à sua seção de choque.

Dentre as 4 interações implementadas no código CUBMC, a única que não apresenta simplificações (quando comparada ao código PENELOPE) é o Espalhamento Compton. A rotina que simula o Efeito Compton presente no código CUBMC é a mesma presente no código PENELOPE. As únicas alterações feitas foram devido à diferença de linguagem, pois o PENELOPE [5] é um código escrito em fortran [9], enquanto o código CUBMC é escrito em CUDA [29], que se baseia em C/C++. No entanto, a estrutura do algoritmo permanece a mesma.

As demais interações foram inicialmente simplificadas, mas estão em processo de modificação para contemplar modelos físicos mais completos como, por exemplo, incluindo uma distribuição de probabilidade para o ângulo de espalhamento na simulação do espalhamento Rayleigh, e relaxamento atômico no efeito fotoelétrico e espalhamento compton, ainda tomando como referência o código PENELOPE.

No espalhamento Rayleigh, existe uma distribuição de probabilidade para a direção do fóton emergente. A aproximação atual adota uma distribuição isotrópica.

No Efeito Fotoelétrico e na Produção de Pares, o modelo atual deposita toda a energia do fóton localmente e encerra sua história. Essa aproximação se justifica pelo livre caminho médio do fóton ser ordens de grandeza maior que o do elétron, fazendo com que a probabilidade do elétron abandonar o voxel seja baixa. No entanto, ignora a probabilidade de ocorrência de bremsstrahlung (efeito no qual um elétron é absorvido pelo material e ejeta um fóton), que é significativo em baixas energias. Essas aproximações serão corrigidas nas próximas versões do código CUBMC. No momento, o código tem seus testes realizados nos regimes em que o Efeito Compton é dominante.

5.3.8 Estimadores (Tallies)

Os ‘tallies’ correspondem às grandezas macroscópicas a serem obtidas como resultado da simulação. Basicamente, são contadores que armazenam as informações de interesse como fluxo ou dose nos voxels de interesse. Por se tratarem de dados obtidos estatisticamente, existe um erro estatístico intrínseco que também deve estar presente.

Atualmente, existem dois ‘tallies’ presentes no código CUBMC, o primeiro deles é o de cálculo de dose, que consiste em armazenar a informação a respeito da

deposição de energia nos voxels, e o segundo ‘tally’ é referente às interações ocorridas, quantas interações de cada tipo ocorreram e qual a energia por elas depositada. Este segundo ‘tally’ é importante durante o desenvolvimento do código, pois permite estimar a participação de cada interação na deposição de energia total do sistema.

5.3.9 Finalização da história da partícula

A história da partícula é então descrita por uma sucessão de passos seguidos por interações até ser finalizada. Existem três condições que implicam na finalização da história da partícula. Ela abandonar a geometria do problema, atingir a energia de corte, ou atingir um valor limite de ‘loops’ no contador. A figura 5.13 mostra o trecho de código no qual essas condições estão explícitas.

```
while(X<Xmax && X>Xmin && Y<Ymax && Y>Ymin && Z<Zmax && Z>Zmin && j< LimParcial && E>Emin){
```

Figura 5.13: Trecho do código CUBMC que define até quando a história de uma partícula é simulada

As duas primeiras condições são estabelecidas devido ao seu significado físico. A história de uma partícula fora da geometria de interesse não é relevante para a simulação, e a energia de corte define o limiar inferior onde a energia depositada pela partícula é desprezível estatisticamente, ou os modelos físicos adotados não são mais válidos.

O limite no contador de ‘loops’ não possui significado físico, ele é apenas um processo preventivo para o caso de alguma partícula ficar em um loop infinito (muito mais demorado que os outros) e resultar em um atraso na simulação. O valor limite adotado é de 10^6 , sendo que um fóton raramente atinge a marca de 100 ‘loops’ antes de atingir sua energia de corte. O valor limite não interfere no resultado da simulação, apenas previne atrasos globais em decorrência de histórias pontuais de significância estatística desprezível.

5.3.10 Método de Woodcock

O código CUBMC pode realizar a simulação da história da partícula usando o método convencional, que é o método de paradas, ou o Método de Woodcock, que foi

descrito na seção 4.2.3.2.

A estrutura geral da simulação da história da partícula é praticamente a mesma, mas sem o módulo de colisão com as fronteiras. A partícula tem seu passo sorteado e se desloca ignorando qualquer fronteira do material.

Para o sorteio do passo, utiliza-se o valor da maior seção de choque macroscópica entre os materiais, que pode ser obtida de duas formas. A primeira delas consiste em ler a tabela de todos os materiais durante a execução do *kernel*, e escolher a de maior valor. A segunda forma seria realizar essa leitura fora do *kernel*, e gerar uma nova tabela contendo o valor de seção de choque a ser usada em cada energia. A vantagem do segundo método, é que, uma vez que a tabela seja gerada, a execução no *kernel* seria mais rápida, pois apenas uma tabela seria lida. No entanto, esse procedimento implicaria em se alocar mais uma tabela na memória constante da GPU, e isso não seria possível pois a memória constante tem apenas 64KB. Nas próximas versões do CUBMC, as tabelas de seção de choque serão alocadas na memória de textura e existirá uma tabela contendo apenas os valores de seção de choque máxima nas simulações usando o Método de Woodcock. No modelo atual, todas as tabelas são lidas e o maior valor é escolhido, conforme vemos no trecho de código presente na figura 5.14.

Com o valor de ST (seção de choque macroscópica) obtido, o passo é realizado ignorando a existência das fronteiras. No entanto, é introduzida uma probabilidade de que ocorra uma interação após o passo. Caso não ocorra interação, um novo passo é sorteado partindo da posição inicial. Esse processo é repetido até que ocorra uma interação.

A probabilidade de ocorrência de uma interação é dada pela razão entre a seção de choque macroscópica do voxel no qual a partícula se encontra após o passo, e a seção de choque macroscópica máxima (usada para o sorteio do passo).

Caso seja definido que ocorrerá uma interação, o processo se segue para a rotina das interações e ‘tallies’ conforme o método de paradas.

5.3.11 Simulações

As simulações de teste foram realizadas para quatro objetos simuladores distintos, compostos por água, osso, pulmão e osso+vácuo. Em cada um dos casos, a geometria

```

#ifdef _WOODCOCK_
Interacao=0;

ID= floor(X)+floor(Y)*DimX+floor(Z)*DimX*DimY;

ST=0e0f;
    for (int mat = 0; mat<NUM_MAT; mat++){
        for(i=DivMat[mat]-1; i>=0;i--){
            if(Energia[mat][i]<E){
                KE=i+1;
                break;
            }
        }
        P1i=SGRA[mat][KE];
        P2i=SGC0[mat][KE];
        P3i=SGPH[mat][KE];
        P4i=SGPP[mat][KE];
        STi[mat]=P1i+P2i+P3i+P4i;
        if(STi[mat]>ST){
            ST=STi[mat];
            P1=P1i;
            P2=P2i;
            P3=P3i;
            P4=P4i;
        }
    }
}

```

Figura 5.14: Trecho do código CUBMC para se obter o maior valor de seção de choque macroscópica entre os diferentes materiais

do problema foi descrita por um objeto simulador cúbico de $64 \times 64 \times 64 \text{ cm}^3$, composto por $128 \times 128 \times 128$ voxels, cada voxel com $0.5 \times 0.5 \times 0.5 \text{ cm}^3$.

Foi simulada uma fonte de fóton pontual de 6 MeV posicionada em $X = Y = Z = 32.1 \text{ cm}$, contida no voxel de índice (64,64,64), posicionada ligeiramente deslocada do centro da geometria para não ser definida na fronteira entre voxels, pois tal posicionamento acarretaria em problemas para se realizar as mesmas simulações nos códigos de referência.

As simulações foram realizadas através do Método de Paradas e de Woodcock e o valor da energia depositada nos voxels foi comparado com simulações realizadas com os códigos MCNP5 e PENELOPE. O tempo total de execução das simulações no código CUBMC foi comparado ao tempo de execução das mesmas simulações no código PENELOPE para se estimar o ganho de eficiência.

Já para comparativos de valor de dose, foram usadas como referência as simulações realizadas com o código MCNP5. Essa escolha foi feita devido ao fato de termos acesso a um *cluster* para realizar as simulações em MCNP5. Dessa forma, foi possível simular um maior número de partículas de forma a minimizar o erro estatístico e obter um melhor comparativo de dose.

Embora não exista nenhuma fronteira física real em um material homogêneo, a geometria do problema é descrita por voxels, e essas fronteiras entre voxels obrigam as partículas a realizarem uma parada para consulta de material quando a simulação é realizada pelo método convencional. Isso gera um aumento significativo no tempo de execução, pois uma partícula pode cruzar centenas de fronteiras antes de ter sua história encerrada.

As simulações realizadas usando o código PENELOPE foram executadas em um processador Intel Core(TM) i7-3520M CPU 2.90GHz, enquanto que as simulações com o MCNP5 (Com exceção do caso da água, que foi simulado em um processador Intel Xeon E5645) foram executadas em um *cluster* SGI Altix XE 340 (composto por 96 processadores Intel Xeon six-core 5650 2.66 GHz), e as simulações do código CUBMC foram executadas em uma GPU GTX560Ti, da Nvidia. Para comparação de dose, foi escolhida uma coluna de voxels passando acima da fonte contida no plano médio entre a fonte e a fronteira da geometria. A coluna foi definida pelos voxels de índice $Vx = [0 : 127]$, $Vy = 64$ e $Vz = 96$, conforme ilustra a figura 5.15

Para um comparativo entre os tempos de execução, foram simuladas 2×10^8 partículas nos códigos PENELOPE e CUBMC. A tabela 5.5 apresenta os tempos de execução envolvidos nas simulações e o ganho de desempenho obtido pelas simulações em CUBMC frente as simulações realizadas no código PENELOPE.

Podemos observar um significativo ganho de desempenho. No entanto, é necessário verificar se o valor de dose calculado está correto. Este comparativo não pode ser feito de forma adequada para uma simulação contendo apenas 2×10^8 partículas, pois a flutuação estatística ainda é muito alta. Para um maior número de partículas, as simulações no PENELOPE levariam muito tempo (semanas). Desse modo, o código MCNP5 foi usado para comparativo de dose.

A figura 5.16 mostra um comparativo de dose para as simulações em objeto

Figura 5.15: Modelo fora de escala da posição da coluna de voxels (em azul) passando acima do voxel da fonte (em vermelho)

Tabela 5.5: Comparativo entre os tempos de execução para as simulações realizadas com o código PENELOPE e CUBMC

PENELOPE		CUBMC			
		Woodcock	Ganho	Paradas	Ganho
ÁGUA	10051s	121s	83x	751s	13x
PULMÃO	9839s	210s	47x	684s	14x
OSSO	36030s	478s	75x	2349s	15x

simulador de água realizadas pelo CUBMC pelo método de paradas e de Woodcock, com a simulação realizada pelo MCNP5. Nas simulações realizadas através do código CUBMC, foram transportadas 10^{10} partículas, enquanto no MCNP5 foram simuladas 4×10^8 . Dessa forma, foi possível obter incertezas estatísticas inferiores a 1.5% nas simulações de ambos

os códigos.

A simulação em MCNP5 levou 18h31min para ser realizada em um processador Intel Xeon E5645 de 24 núcleos. Já as simulações com o CUBMC levaram 1h15min com o método de Woodcock e 7h38min com o método de paradas.

Figura 5.16: Comparativo de doses para a simulação do transporte de fótons em um meio de água. Foram simulados 10^{10} fótons pelo código CUBMC e 4×10^8 pelo MCNP5

Podemos observar na figura 5.16 uma boa concordância entre as diferentes simulações. Fazendo um comparativo das doses calculadas pelo CUBMC e MCNP5, voxel a voxel, obtemos um desvio máximo de 8%, com a média aritmética dos desvios sendo inferior a 2%. O mesmo comportamento foi observado nas simulações para os casos homogêneos de pulmão e osso, conforme mostram as figuras 5.17 e 5.18, respectivamente.

A simulação em MCNP5, dessa vez realizada no *cluster* SGI Altix XE 340, levou 3h40min no caso do pulmão e 4h02min para o caso de osso, enquanto as simulações em CUBMC levaram, no caso do pulmão, 2h15min com o método de Woodcock e 7h16min com o método de paradas, e no caso do osso, 5h07min com o método de Woodcock e 22h28min com o método de paradas. Conforme mostra a tabela 5.6

Para se estabelecer uma comparação quantitativa entre os cálculos de dose

Figura 5.17: Comparativo de doses para a simulação do transporte de fótons em um meio composto por pulmão. Foram simulados 10^{10} fótons pelo código CUBMC e 4×10^8 pelo MCNP5

Tabela 5.6: Comparativo entre os tempos de execução para as simulações realizadas com o código MCNP5 e CUBMC

	MCNP5		CUBMC		
		Woodcock	Ganho	Paradas	Ganho
ÁGUA ^a	18h31m	1h15m	14.81x	7h38m	2.43x
PULMÃO	3h40m	2h15m	1.63x	7h16m	0.50x
OSSO	4h02m	5h07m	0.79x	22h28m	0.18x

^aSimulação não realizada no *cluster*

obtidos pelo MCNP5 e CUBMC, as doses foram comparadas voxel a voxel, assim, temos o valor de desvio máximo obtido, e o desvio médio (média aritmética dos desvios). Esse método foi aplicado para os 128 voxels apresentados nas figuras anteriores, e os valores obtidos de desvio de dose estão na tabela 5.7.

Figura 5.18: Comparativo de doses para a simulação do transporte de fótons em um meio composto por osso. Foram simulados 10^{10} fótons pelo código CUBMC e 4×10^8 pelo MCNP5

Tabela 5.7: Comparativo entre os valores de doses obtidos nas simulações realizadas com o código MCNP5 e CUBMC

Meio	Woodcock		Paradas	
	Max	Med	Max	Med
Água	7%	1.87%	8%	1.89%
Pulmão	11%	2.89%	9%	2.75%
Osso	14%	3.22%	14%	3.21%

O quarto caso analisado foi o de distribuição de dose em um objeto simulador heterogêneo, composto por fatias intercaladas de água-vácuo-água-vácuo-água. As fatias de vácuo estão entre os planos $V_x=[46:48]$ e $V_x=[52:54]$. Nesse caso, para termos uma melhor precisão próximo às fronteiras entre os materiais, foram simuladas 10^{11} partículas no CUBMC, atingindo assim, erro estatístico inferior a 0.3%. A figura 5.19 mostra a distribuição de dose para as simulações realizadas no código CUBMC e MCNP5.

Figura 5.19: Comparativo de doses para a simulação do transporte de fótons em um objeto simulador heterogêneo, composto por fatias intercaladas de água-vácuo-água-vácuo-água. Foram simulados 10^{11} fótons pelo código CUBMC e 4×10^8 pelo MCNP5

Como podemos observar, existe uma boa concordância geral entre os resultados da simulação em MCNP e CUBMC. O desvio médio obtido com método de paradas foi de 1,6% e com o Método de Woodcock foi de 3,8%. No entanto, ambos os métodos apresentaram um desvio mais significativo nos voxels próximos às fronteiras entre os materiais, atingindo o desvio máximo de 4% pelo método de paradas e 8% pelo método de Woodcock.

6 Discussões

O código CUBMC ainda está em processo de desenvolvimento mas, até o momento, tem demonstrado estar em boa concordância com os códigos de referência, PENELOPE e MCNP5. Ele ainda apresenta várias simplificações que serão corrigidas em versões futuras do código, envolvendo modelos físicos mais completos e abrangentes.

De maneira geral, houve um ganho de performance significativo (tabela 5.7) das simulações executadas com o código CUBMC quando comparadas com as simulações realizadas com o código PENELOPE, mas foi observado que esse ganho é dependente do material do meio do objeto simulador. Na simulação com ganho de desempenho menos significativo, este foi de 50x com o método de Woodcock e de aproximadamente 10x com o método de paradas. Embora as simulações tenham sido executadas em processadores diferentes, isso não é significativo para as comparações, pois uma simulação no código CUBMC tem mais de 99% de seu processamento realizado pela GPU. Uma simulação que demora cerca de 3 horas para um processador I7 executando o código PENELOPE, demora menos de 3 minutos quando executada no código CUBMC em uma unidade de processamento gráfico. Vale salientar que a unidade de processamento gráfico usada (GTX560Ti) é uma GPU modesta comparada a outras linhas de GPUs da Nvidia, ou modelos atuais da linha GTX, podendo ser considerada uma GPU de baixo custo.

As diferenças médias de dose entre as simulações em CUBMC e MCNP5 foram inferiores a 3.5% nos testes comparativos já realizados. Comparando-se as doses voxel a voxel, as diferenças máximas obtidas foram de 7% para o objeto simulador de água, 11% para o de pulmão e 14% para o de osso. Parte dessas diferenças é devido às simplificações adotadas nos modelos físicos usados no código CUBMC, e devem ser reduzidas quando modelos mais completos forem adotados. Outro motivo conhecido para que exista essa diferença é o fato dos códigos trabalharem com tabelas de seção de choque diferentes.

Embora a tabela 5.6 apresente um comparativo entre os tempos de simulação com o MCNP5 e CUBMC, esses valores ainda devem ser obtidos de maneira mais criteriosa. Dois fatores relevantes devem ser considerados. O primeiro deles é que, não sabe-

mos quantos processadores do *cluster* SGI foram utilizados. Como o *cluster* pertence ao grupo de pesquisa do CEN-IPEN (Centro de Engenharia Nuclear - Instituto de Pesquisas Energéticas e Nucleares), ele poderia estar realizando simulações de outros pesquisadores, e sua capacidade de processamento não foi destinada exclusivamente às simulações apresentadas neste trabalho. Um segundo fator importante é a relação entre o número de partículas simuladas entre os dois códigos (4×10^8 no MCNP5 e 10^{10} no CUBMC). Foi observado que o código MCNP5 apresenta uma menor flutuação estatística para simulações com o mesmo número de partículas que simulações realizadas no código PENELOPE ou CUBMC. Dessa forma, um maior número de partículas foi simulado pelo CUBMC, no entanto, esse número foi definido de maneira empírica, sendo maior do que o número necessário para obter a mesma precisão estatística que o MCNP5. Podemos ver na figura 5.18 que a flutuação estatística da simulação com o MCNP5 é maior que a das simulações no CUBMC.

Ainda assim, foi observado que o tempo de execução das simulações em CUBMC pelo método de Woodcock são comparáveis aos tempos das simulações de MCNP5 no *cluster*, o que é um dado significativo, principalmente se levarmos em consideração que o *cluster* é composto por 96 processadores Xeon e as simulações em CUBMC foram executadas em uma única GPU (GTX560Ti). Testes também apresentaram que simulações executadas na GPU GTX770m (de notebook) apresentaram um tempo apenas 20% superior que o tempo da GPU GTX560Ti (de desktop), o que significa que temos a possibilidade de realizar simulações em um notebook com tempos de execução comparáveis ao tempo das simulações em um *cluster*.

Ao se introduzir um objeto simulador heterogêneo contendo vácuo, obtemos boa concordância global entre as simulações, mas divergências pontuais que devem ser averiguadas. Enquanto a simulação realizada pelo método de paradas apresenta uma melhor concordância global com a simulação realizada pelo MCNP, ela superestima a dose próximo às fronteiras entre os materiais. Tal efeito é intensificado nas simulações realizadas pelo Método de Woodcock. O motivo dessas divergências está sendo estudado.

7 Conclusão

O código CUBMC (CUDA Based Monte Carlo) é um código de Monte Carlo destinado ao transporte de fótons em objetos simuladores definidos por voxels. O objetivo é que ele seja uma opção de código simulador que, ao utilizar a capacidade de processamento paralelo de unidades de processamento gráfico (GPU), apresente alto desempenho em máquinas compactas e de baixo custo, podendo assim ser aplicado em casos clínicos e incorporado a sistemas de planejamento.

Dessa forma, a descrição da geometria por voxels busca facilitar a descrição de modelos anatômicos provenientes de imagens de Tomografia Computadorizada (TC) ou Ressonância Magnética (RM), que são facilmente extrapoladas para voxels. Visando obter ganho de desempenho, a física do transporte de radiação presente no código foi simplificada ao máximo dentro do limite em que ainda apresente um resultado adequado no intervalo energético de interesse para os tratamentos de radioterapia e braquiterapia (1keV - 9MeV).

O gerenciamento da memória da GPU também foi realizado de modo a se usar, prioritariamente, as memórias de rápido acesso (memória de registro e constante) e minimizar o tráfego na memória global (mais lenta). Essa opção fez com que fosse possível descrever a geometria com até 3 materiais diferentes, o que é um número muito restritivo para objetos simuladores baseados em modelos anatômicos reais. Em versões futuras do CUBMC, a alocação de memória deverá ser alterada de modo a possibilitar o aumento do número de materiais compondo a geometria, mesmo que em detrimento do desempenho das simulações.

O CUBMC se restringe a transportar fótons, as partículas secundárias geradas nas interações não são transportadas e sua energia é depositada localmente. Dentre as interações simuladas (Efeito Fotoelétrico, Espalhamento Rayleigh, Espalhamento Compton e Produção de Pares), apenas o Espalhamento Compton apresenta um algoritmo de simulação mais abrangente. Ele foi extraído sem alterações do código PENELOPE. As outras interações são extremamente simplificadas e serão implementadas à medida que o

código CUBMC for aprimorado.

O código CUBMC permite que o transporte dos fótons seja simulado segundo dois métodos. O método convencional em que a partícula é obrigada a realizar uma parada toda vez que encontrar um fronteira da geometria (chamado neste trabalho de ‘método de paradas’), e o método em que a partícula é transportada em um meio homogêneo fictício, ignorando as fronteiras da geometria (Método de Woodcock).

Mesmo com o processo de simulação das interações simplificado, simulações realizadas no CUBMC para meios homogêneos com fonte pontual monoenergética de 6MeV apresentaram desvios médios de dose inferiores a 3.5% quando comparadas ao MCNP5.

Estabelecendo-se um comparativo entre os tempos de simulação, as simulações em CUBMC (sempre executadas na GPU GTX560Ti) segundo o método de paradas foram, aproximadamente, 15 vezes mais rápidas que as mesmas simulações executadas no código PENELOPE (executado em um processador Intel Core i7-3520M de 2.9.GHz). Já pelo Método de Woodcock, as simulações foram, em média, 70 vezes mais rápidas.

As simulações realizadas com o código MCNP5 foram executadas em um *cluster* composto por 96 processadores Intel Xeon six-core 5650. No entanto, o objetivo dessas simulações era para se estabelecer um comparativo de dose, e não de tempo de execução. Dessa forma, não foi controlado se o *cluster* teve toda a sua capacidade de processamento dedicada às simulações, e o número de partículas simuladas pelo CUBMC foi superior ao necessário para se obter uma flutuação estatística semelhante a obtida pelo MCNP5. Sendo assim, os tempos de execução obtidos servem apenas como parâmetro de referência, mas não devem ser tomados como valores corretos. Tendo essa ressalva, o tempo de execução para as simulações em CUBMC pelo Método de Woodcock foi próximo ao tempo obtido pelo *cluster*, e as simulações pelo método de paradas foram de 2 a 5 vezes mais lentas.

Embora o comparativo de tempo não tenha sido estabelecido de forma criteriosa, o fato dos tempos de execução serem próximos, e em alguns casos o CUBMC ter sido mais rápido que o *cluster*, nos mostra o grande potencial presente no código CUBMC, pois obteve, em uma única GPU, um desempenho comparável ao de um *cluster* composto por 96 processadores executando o MCNP5.

O código CUBMC ainda precisa ser aprimorado para alcançar resultados mais precisos nas simulações em meios heterogêneos, devem ser implementados modelos físicos mais detalhados das interações dos fótons para se obter cálculos de dose mais realistas. Embora ainda seja aplicável apenas a casos restritos, as simulações realizadas com o código CUBMC apresentaram uma boa concordância com os códigos de referência e com um significativo ganho de desempenho.

Os resultados aqui apresentados demonstram o potencial do código CUBMC de se tornar, em um futuro próximo, uma opção para simular o transporte de fótons pelo Método de Monte Carlo de forma rápida. Ao se tornar um código mais abrangente, capaz de simular o transporte em objetos simuladores provenientes de modelos anatômicos reais, terá utilização prática em sistemas de planejamento de tratamento em radioterapia.

Referências Bibliográficas

- [1] E. Podgorsak *et al.*, “Radiation oncology physics,” *a handbook for teachers and students/EB Podgorsak*.—Vienna: International Atomic Energy Agency, vol. 657, 2005.
- [2] International Atomic Energy Agency, “Absorbed dose determination in external beam radiotherapy,” *Technical Report Series No. 398*. Vienna: IAEA, 2000.
- [3] J. Van Dyk and J. Van Dyk, *The Modern Technology of Radiation Oncology: A Compendium for Medical Physicists and Radiation Oncologists*. No. v. 1, Medical Physics Pub., 1999.
- [4] J. T. Goorley, M. R. James, T. E. Booth, F. B. Brown, J. S. Bull, L. J. Cox, J. W. J. Durkee, J. S. Elson, M. L. Fensin, R. A. I. L. A. N. L. Forster, and et al., *Initial MCNP6 Release Overview - MCNP6 Beta 3*. Nov 2012.
- [5] J. Baro, J. Sempau, J. Fernandez-Varea, and F. Salvat, “Penelope: An algorithm for monte carlo simulation of the penetration and energy loss of electrons and positrons in matter,” *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms*, vol. 100, no. 1, pp. 31 – 46, 1995.
- [6] Geant4 Collaboration, “Geant4 user’s guide for application developers,” Dezember 2009.
- [7] A. Badal and J. Sempau, “A package of linux scripts for the parallelization of monte carlo simulations,” *Computer Physics Communications*, vol. 175, no. 6, pp. 440 – 450, 2006.
- [8] B. Stroustrup *et al.*, *The C++ programming language*. Pearson Education India, 1995.
- [9] D. J. Veldman, *Fortran programming for the behavioral sciences*. Holt, Rinehart and Winston New York, 1967.

- [10] J. Leppänen, “Serpent—a continuous-energy monte carlo reactor physics burnup calculation code,” *VTT Technical Research Centre of Finland*, 2012.
- [11] N. Metropolis and S. Ulam, “The monte carlo method,” *Journal of the American statistical association*, vol. 44, no. 247, pp. 335–341, 1949.
- [12] D. Rogers, “Fifty years of monte carlo simulations for medical physics,” *Physics in medicine and biology*, vol. 51, no. 13, p. R287, 2006.
- [13] P. Andreo, “Monte carlo techniques in medical radiation physics,” *Physics in Medicine and Biology*, vol. 36, no. 7, p. 861, 1991.
- [14] I. Kawrakow, “Vmc++, electron and photon monte carlo calculations optimized for radiation treatment planning,” in *Advanced Monte Carlo for Radiation Physics, Particle Transport Simulation and Applications.*, vol. 1, p. 229, 2001.
- [15] I. Kawrakow and D. Rogers, “The egsrc code system: Monte carlo simulation of electron and photon transport,” 2000.
- [16] J. Sempau, S. J. Wilderman, and A. F. Bielajew, “Dpm, a fast, accurate monte carlo code optimized for photon and electron radiotherapy treatment planning dose calculations,” *Physics in medicine and biology*, vol. 45, no. 8, p. 2263, 2000.
- [17] C.-M. Ma, J. Li, T. Pawlicki, S. Jiang, and J. Deng, “Mcdose â a monte carlo dose calculation tool for radiation therapy treatment planning,” in *The Use of Computers in Radiation Therapy*, pp. 123–125, Springer, 2000.
- [18] R. Kramer, J. Vieira, H. Khoury, F. Lima, and D. Fuelle, “All about max: a male adult voxel phantom for monte carlo calculations in radiation protection dosimetry,” *Physics in medicine and biology*, vol. 48, no. 10, p. 1239, 2003.
- [19] R. Kramer, H. Khoury, J. Vieira, E. Loureiro, V. Lima, F. Lima, and G. Hoff, “All about fax: a female adult voxel phantom for monte carlo calculation in radiation protection dosimetry,” *Physics in medicine and biology*, vol. 49, no. 23, p. 5203, 2004.

- [20] G. Bueno, O. Déniz, C. Carrascosa, J. Delgado, and L. Brualla, “Fast monte carlo simulation on a voxelized human phantom deformed to a patient,” *Medical physics*, vol. 36, no. 11, pp. 5162–5174, 2009.
- [21] R. Cruise, R. Sheppard, and V. Moskvina, “Parallelization of the penelope monte carlo particle transport simulation package,” *Nuclear Mathematical and Computational Sciences: A Century in Review, A Century Anew*, pp. 6–11, 2003.
- [22] F. B. Brown, R. Barrett, T. Booth, J. Bull, L. Cox, R. Forster, T. Goorley, R. Mosteller, S. Post, R. Prael, *et al.*, “Mcnp version 5,” *Trans. Am. Nucl. Soc.*, vol. 87, no. 4, 2002.
- [23] G. Pratz and L. Xing, “Gpu computing in medical physics: A review,” *Medical physics*, vol. 38, no. 5, pp. 2685–2697, 2011.
- [24] A. Badal and A. Badano, “Accelerating monte carlo simulations of photon transport in a voxelized geometry using a massively parallel graphics processing unit,” *Medical physics*, vol. 36, no. 11, pp. 4878–4880, 2009.
- [25] X. Jia, X. Gu, J. Sempau, D. Choi, A. Majumdar, and S. B. Jiang, “Development of a gpu-based monte carlo dose calculation code for coupled electron–photon transport,” *Physics in medicine and biology*, vol. 55, no. 11, p. 3077, 2010.
- [26] S. Hissoiny, B. Ozell, H. Bouchard, and P. Després, “Gpumcd: A new gpu-oriented monte carlo dose calculation platform,” *Medical physics*, vol. 38, no. 2, pp. 754–764, 2011.
- [27] E. Alerstam, T. Svensson, and S. Andersson-Engels, “Parallel computing with graphics processing units for high-speed monte carlo simulation of photon migration,” *Journal of biomedical optics*, vol. 13, no. 6, pp. 060504–060504, 2008.
- [28] X. Jia, H. Yan, X. Gu, and S. B. Jiang, “Fast monte carlo simulation for patient-specific ct/cbct imaging dose calculation,” *Physics in medicine and biology*, vol. 57, no. 3, p. 577, 2012.
- [29] NVIDIA Corporation, *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2007.

- [30] I. G. Zubal, C. R. Harrell, E. O. Smith, Z. Rattner, G. Gindi, and P. B. Hoffer, “Computerized three-dimensional segmented human anatomy,” *Medical physics*, vol. 21, no. 2, pp. 299–302, 1994.
- [31] P. L’Ecuyer, “Efficient and portable combined random number generators,” *Commun. ACM*, vol. 31, pp. 742–751, June 1988.
- [32] F. Salvat, J. M. Fernández-Varea, and J. Sempau, “Penelope-2006: A code system for monte carlo simulation of electron and photon transport,” in *Workshop Proceedings*, vol. 4, p. 7, 2006.
- [33] M. J. Berger, J. Hubbell, S. Seltzer, J. Chang, J. Coursey, R. Sukumar, D. Zucker, and K. Olsen, “Xcom: photon cross sections database,” *NIST Standard reference database*, vol. 8, pp. 87–3597, 1998.
- [34] E. Woodcock, T. Murphy, P. Hemmings, and S. Longworth, “Techniques used in the gem code for monte carlo neutronics calculations in reactors and other systems of complex geometry,” in *Proc. Conf. Applications of Computing Methods to Reactor Problems*, vol. 557, 1965.
- [35] O. Klein and Y. Nishina, “On the scattering of radiation by free electrons according to dirac’s new relativistic quantum dynamics,” *The Oskar Klein Memorial Lectures: 1988-1999*, vol. 1, pp. 253–272, 2014.
- [36] G. P. Fonseca, P.C.G. Antunes, B. Reniers, H. Yoriyaz, and F. Verhaegen, “A brachytherapy model-based dose calculation algorithm - amigobrachy,” *International Nuclear Atlantic Conference INAC 2013*, 2013.
- [37] The mathworks, “Matlab user’s guide,” *Inc., Natick, MA*, vol. 5, 1998.
- [38] National Electrical Manufacturers Association and American College of Radiology and others, *Digital imaging and communications in medicine (DICOM)*. National Electrical Manufacturers Association, 1998.
- [39] Oncentra, Nucletron, “Masterplan,” *Physics and algorithms manual (192.739 ENG-01)*. Nucletron Pty. Ltd. The Netherlands.

-
- [40] Hibbett, Karlsson, and Sorensen, *ABAQUS/standard: User's Manual*, vol. 1. Hibbett, Karlsson & Sorensen, 1998.