RICARDO CÉSAR RIBEIRO DOS SANTOS

RECONSTRUÇÃO DIGITAL MASSIVAMENTE PARALELA DE CORPOS DE PROVA DE CONCRETO A PARTIR DE TOMOGRAFIA INDUSTRIAL

CURITIBA, 2014

RICARDO CÉSAR RIBEIRO DOS SANTOS

RECONSTRUÇÃO DIGITAL MASSIVAMENTE PARALELA DE CORPOS DE PROVA DE CONCRETO A PARTIR DE TOMOGRAFIA INDUSTRIAL

Dissertação apresentada ao Programa de Pós-Graduação em Métodos Numéricos em Engenharia, área de concentração Mecânica Computacional, Setor de Tecnologia, Departamento de Construção Civil e Setor de Ciências Exatas, Departamento de Matemática da Universidade Federal do Paraná, como requisito parcial para a obtenção do título de Mestre em Ciências.

Orientador: Prof. Dr. Klaus de Geus

Co-Orientador: Prof. Dr. Walmor Cardoso Godoi

CURITIBA, 2014

TERMO DE APROVAÇÃO

RICARDO CÉSAR RIBEIRO DOS SANTOS

RECONSTRUÇÃO DIGITAL MASSIVAMENTE PARALELA DE CORPOS DE PROVA DE CONCRETO A PARTIR DE TOMOGRAFIA INDUSTRIAL

Dissertação aprovada como requisito parcial para a obtenção do grau de Mestre no Curso de Pós-Graduação em Métodos Numéricos em Engenharia, área de concentração Mecânica Computacional, Setores de Tecnologia e de Ciências Exatas da Universidade Federal do Paraná, pela seguinte banca examinadora:

Prof. Klaus de Geus

Programa de Pós-Graduação em Métodos Numéricos em Engenharia – PPGME/UFPR

Prof. Sergio Scheer

Programa de Pós-Graduação em Métodos Numéricos em Engenharia – PPGME/UFPR

Prof. Cinthia Obladen de Almendra Freitas

Programa de Pós-Graduação em Informática – PPGIa/PUCPR Escola Politécnica

CURITIBA, 2014

À Carol, minha esposa.

Aos amigos, aos parentes, aos meus pais e às minhas irmãs.

A todos que me ajudaram a descobrir a informática e que me ajudaram nos passos até aqui.

AGRADECIMENTOS

A todos que contribuíram para este trabalho, agradeço.

À minha esposa que esteve comigo e sempre teve fé no meu trabalho, agradeço.

A todos que perguntaram sobre o que era o trabalho e me ouviram falar durante o que, com certeza, pareceram horas, agradeço.

Aos colegas de laboratório, agradeço.

Ao meu orientador, agradeço pela sabedoria, pelo tempo, pela paciência e pela aparente fé constante.

Ao meu co-orientador, agradeço pela sabedoria, pelo tempo, pela paciência e por sempre me lembrar que existem as tomografias e o tomógrafo.

Ao professor Sergio Scheer, agradeço.

À Capes, agradeço pelo apoio financeiro.

À ANEEL (Agência Nacional de Energia Elétrica), à COPEL, aos Institutos LACTEC e à UFPR agradeço pela infra-estrutura disponibilizada e pela viabilização do trabalho através do contexto do projeto.

RESUMO

Ensaios não destrutivos são importantes para a análise de estruturas de concreto. Sua realização, entretanto, necessita de amostras que devem ser retiradas das estruturas a serem analisadas, tais como barragens e usinas. A tomografia industrial, mediante a obtenção de mapas de densidade, aliada a modelos e ferramentas de simulação, têm um papel relevante no contexto de ensaios não destrutivos. Como as ferramentas de simulação atuais fazem uso de modelos geométricos poligonais, torna-se necessário extrair superfícies mediante a geração de malhas poligonais, a partir do volume de dados, usando técnicas tais como o algoritmo Marching Cubes. Este trabalho trata da geração de malhas poligonais de alta definição a partir de testemunhos de concreto, os quais apresentam dimensões que normalmente impactam o desempenho, o consumo de memória e até mesmo a factibilidade do processamento. Para a solução destes desafios, foi contemplada a solução utilizando uma implementação massivamente paralela do algoritmo Marching Cubes. Foi implementada uma versão paralela deste algoritmo utilizando a linguagem CUDA/C para programação da GPU de forma a conseguir ganho de tempo de processamento na geração de malhas de alta definição para imagens tomográficas de corpos de prova de concreto. A avaliação do ganho de velocidade de processamento foi realizada comparando-se a solução paralelizada com um protótipo desenvolvido em MATLAB. Foram desenvolvidos módulos de software em código protótipo para auxiliar na análise da solução. As análises realizadas mostram que o uso de paralelismo massivo mediante GPU tem grande potencial para esse tipo de aplicação, não obstante o aumento no consumo de memória devido à necessidade de controle mais sofisticado na transferência de dados no processamento. A redução do tempo de processamento obtida como resultado neste estudo foi significativa, a saber, de doze horas na arquitetura serial para pouco mais de quarenta segundos na arquitetura paralela.

Palavras-chave: GPU, CUDA, *Marching Cubes*, Tomografia Industrial, Geração de Isossuperfícies Poligonais, concreto

ABSTRACT

Non-destructive testing is an important procedure for the analysis of concrete structures. It requires, however, structural samples to be extracted from buildings such as dams or bridges. In order to acquire data regarding the geometry and the topography of both surface and internal structures, industrial tomography can be used, through the acquisition of density maps. This data can be used with simulation software applications and models. However, the aforementioned procedures require polygonal meshes, hence, the necessity to extract surfaces from the volumetric data. In order to create such models, algorithms must be used, such as the Marching Cubes algorithm, with the purpose of generating a virtual model corresponding to the real object in a given density. The purpose of this document is to detail the creation of high density polygonal meshes from concrete samples that generate density maps with dimensions that pose a challenge to both processing time and memory consumption, posing a problem to the feasibility of the solution. To overcome these challenges, the CUDA/C language was used for programming a Graphics Processing Unit (GPU) to decrease the processing time required to extract polygonal surfaces from tomographic images. To evaluate the speed increment of the algorithm, the execution time of the parallelized version was compared with the initial serial MATLAB implementation. Additionally, auxiliary code was implemented aiming a robust memory management. The results confirm that massive parallel GPU programming can be used to reduce the processing time, even though memory consumption, due to the more sophisticated management techniques implemented for data transfer, was higher. As an overall result, the processing time decreased from approximately twelve hours on the serial implementation to approximately forty seconds on the massive parallel approach, with a corresponding increase in memory consumption.

Keywords: GPU, CUDA, Marching Cubes, Industrial Tomography, Polygonal Isossurfaces generation

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CPU – *Central Processing Unit*

GPU – *Graphics Processing Unit*

GPGPU – General Purpose computation on Graphics Processing Unit

CUDA – *Compute Unified Device Architecture*

FLOPS – *FLoating-points Operations Per Second*

TFLOP – Tera (10^{12}) *FLoating-points Operations Per Second*

VTK – *Visualization ToolKit*

P&D – Pesquisa e Desenvolvimento

ANEEL – Agência Nacional de Energia ELétrica

SMX – *Streaming Multiprocessor*

Programação COM – Programação *Component Object Model*

CLR – *Common Language Runtime*

CTS – *Common Type System*

CLS – *Common Language Specification*

CIL – *Common Intermediate Language*

VES – *Virtual Execution System*

Compilador JIT – Compilador *Just In Time*

TIFF – *Tag Image File Format*

ENDs – Ensaios Não Destrutivos

SUMÁRIO

1. INTRODUÇÃO

As estruturas de concreto da engenharia civil são construídas para garantir formas seguras de explorar atividades sociais e econômicas, seja por meio de edifícios de escritórios, barragens hidroelétricas ou torres em que são instaladas antenas para transmissão de dados.

Para garantir a integridade estrutural de uma edificação durante seu ciclo de vida e maximizar sua vida útil, é necessário monitorar frequentemente os materiais que a compõem e verificar sinais de desgaste devido a diferentes fenômenos, tais como: excesso de tensões mecânicas, ação de elementos corrosivos, fenômenos meteorológicos ou ação animal.

Para testar a confiabilidade das estruturas de concreto é necessário retirar amostras dos componentes e ensaiá-los em laboratório, com o inconveniente de que os testes, de forma geral, destroem as amostras, limitando a quantidade de análises possíveis à quantidade de amostras disponíveis.

Uma das alternativas aos testes tradicionais é conduzir ensaios não destrutivos com ferramentas de *software* para simulação mecânica. Porém, para que tais simulações sejam conduzidas, é necessário que as informações relevantes aos testes sejam transportadas a um modelo virtual, que deve se comportar de forma idêntica a como o modelo real se comportaria nas mesmas condições.

Para este fim, são utilizadas técnicas como tomografia industrial tridimensional, impedância elétrica ou ultrassom para discriminar os materiais que compõem uma determinada amostra e catalogá-los de acordo com suas densidades, visando inferir suas características mecânicas. Entretanto, essas tecnologias geram imagens bidimensionais que podem ajudar a diagnosticar patologias por meio da densidade dos materiais. Outras vezes são necessários modelos computacionais instanciados com os parâmetros necessários para a condução de simulações matemáticas em ferramentas de *software* apropriadas.

A fim de elaborar o modelo matemático necessário, utilizam-se as informações

morfológicas e de distribuição de densidade retiradas das fatias tomográficas como entrada para um algoritmo que se encarrega de realizar a criação do modelo computacional que contenha as informações para as simulações necessárias e que seja confiável, comportando-se como o sólido real se comportaria (LORENSEN e CLINE, 1987).

Um dos algoritmos utilizados para este fim é o *Marching Cubes*, elaborado na década de 80 e cuja função é gerar superfícies tridimensionais a partir de imagens tomográficas planas, utilizando os valores de densidade que compõem as superfícies a fim de construir um modelo geométrico tridimensional de alta definição. Uma das desvantagens deste algoritmo é a quantidade de cálculos realizados, já que o algoritmo alinha as imagens e define células constituídas de oito *voxels*[1] adjacentes ao volume resultante. Sendo que o algoritmo deve visitar e catalogar cada uma em sequência. Como o tempo de processamento é proporcional ao volume de dados, observa-se aumento do tempo de processamento para um pequeno aumento no volume de dados para processamento. Por outro lado, o algoritmo foi desenvolvido para ser uma abordagem dividir e conquistar (divide and conquer) (LORENSEN e CLINE, 1987), tornando-o propício a um mecanismo de paralelização (FARBER, 2011).

No caso deste trabalho, a paralelização do algoritmo se baseou nos conceitos de computação paralela heterogênea para diminuir o tempo de execução requerido pelo algoritmo *Marching Cubes* por meio da aplicação da API CUDA (Computer Unified Device Architecture), desenvolvida pela NVIDIA[2].

O método utilizado não só possibilitou o processamento de grandes quantidade de dados, como permitiu manter a geometria da malha gerada o mais próximo possível do sólido original, utilizando todos os dados possíveis de extrair das fatias tomográficas utilizadas como base para o processamento do código elaborado.

Como benefício colateral, este trabalho inaugura a utilização da GPU *(Graphics Processing Unit)* no projeto em que se insere, criando um ambiente de desenvolvimento e teste propício para a continuada implementação de códigos que possam beneficiar ainda

---

1 “Um voxel é um termo utilizado para referenciar um elemento de um volume, como um pixel é um elemento de uma figura e um texel é um elemento de uma textura.” (SHREINER, SELLERS et al., 2013)

2 A NVIDIA é uma empresa americana que se especializa em pesquisa e desenvolvimento de equipamentos para computação gráfica, com uma linha de GPUs para computação visual e de alto desempenho.

mais a área de atuação do laboratório de tomografia industrial.

Por fim, utilizou-se a ferramenta de bibliotecas de visualização denominada Visualization Toolkit (VTK) (KITWARE) para a verificação visual do sólido resultante do processamento do *software*, mostrando as malhas extraídas das tomografias em um ambiente computacional tridimensional.

1.1. PROBLEMÁTICA

O escopo do projeto definia, inicialmente, um fluxo de trabalho que faria com que um corpo de concreto estivesse pronto para ter os dados adquiridos com o uso de um tomógrafo industrial e estas fatias tomográficas fossem utilizadas como base para a reconstrução de um modelo computacional tridimensional para a utilização em ferramentas de simulação mecânica.

Para a criação deste modelo, o algoritmo escolhido foi o *Marching Cubes*. Esta escolha foi influenciada pela sua simplicidade de implementação, pela variedade de implementações já realizadas e por ter bibliografia facilmente disponível sobre o assunto e adaptações realizadas. Outro fator importante na decisão foi o fato que o algoritmo gera uma malha de polígonos que representa as superfícies que delimitam o objeto, entrada necessária para a ferramenta de simulação mecânica.

A simplicidade do algoritmo advém do fato de o *Marching Cubes* ser um algoritmo de força bruta (COOK, 2013). Esta característica faz com que o tempo de execução seja muito elevado e que o consumo de memória seja muito alto também, como é típico neste tipo de algoritmo (SEDGEWICK e FLAJORLET, 2013).

Para a reconstrução do sólido foram utilizadas algumas implementações do algoritmo. Uma delas é a implementação do próprio VTK, mas este apresentava problema de falta de memória durante o cálculo do sólido e abortava a execução. Este erro provavelmente resulta de um gerenciamento de memória deficitário. A saída para este problema foi a simplificação do modelo por meio da redução de polígonos em um fator de redução arbitrário de 95% da quantidade de triângulos do modelo. Este parâmetro em alguns casos chegava a degenerar o sólido de modo a comprometer a utilidade do modelo.

Outra solução para o problema com o VTK é a diminuição do número de fatias

tomográficas a processadas, visando a redução da memória utilizada para armazenamento dos dados. Para que o processamento pudesse ser finalizado, entretanto, a quantidade de fatias a serem processadas deveriam ser de aproximadamente 10 do conjunto de 585. Esta quantidade é insuficiente, uma vez que o sólido deveria ser reconstruído em sua totalidade. Esta abordagem, entretanto, permite apenas a simulação de uma parcela do sólido por vez, impedindo a realização da simulação.

Para eliminar estas limitações foi utilizada a GPU para realizar o processamento maciço paralelo do problema, utilizando a GPU e a memória do *device* com sua capacidade de processamento, liberando a CPU do *host* enquanto o processamento dos dados de entrada é realizado.

Na abordagem da GPU não há eliminação de triângulos, ou seja, sempre é utilizada a definição máxima disponível para o processamento, utilizando todos os dados disponíveis nas imagens tomográficas.

Esta alta definição é desejável não apenas para uma boa visualização do sólido, como também é interessante para ser utilizado como entrada para o aplicativo de simulação mecânica, já que é possível realizar o cálculo preciso do comportamento do sólido quando submetido a determinadas condições ambientais.

1.2. OBJETIVOS

Este trabalho se inseriu no contexto de um projeto P&D ANEEL, finalizado em 2014, que visou a simulação não destrutiva de corpos de prova de concreto e que já teve como produção científica, no assunto diretamente abordado nesta dissertação, o artigo intitulado "Geometry extraction of high resolution CT studies from concrete structural samples: a massive parallel approach using a GPU architecture", publicado no CILAMCE, Ibero-Latin American Congress on Computational Methods in Engineering (SANTOS, GEUS et al., 2014) . O projeto centrou-se nas análises de corpos de prova de concreto retirados da Barragem Hidroelétrica da Derivação do Rio Jordão; realizou ensaios de tomografias e, a partir de dados tridimensionais, realizou simulação mecânica por meio de métodos de elementos finitos, confrontando os resultados com os ensaios destrutivos de laboratório.

Entretanto, as ferramentas adotadas pelo projeto para fazer o tratamento das tomografias até a realização das simulações apresentaram problemas devido ao volume de dados a serem processados. O aplicativo anteriormente utilizado não conseguia processar todo o conjunto de imagens, impedindo a utilização do estudo integral de tomografia nas simulações, o que comprometia o objetivo de todo o processo.

Este trabalho teve como objetivo direto solucionar alguns desses problemas e contribuir com o método desenvolvido no projeto de P&D mediante uma arquitetura mais robusta e eficiente.

Para que o modelo virtual pudesse ser utilizado em conjunto com a ferramenta apropriada para simulação, tornou-se necessário extrair uma isossuperfície a partir do espaço tridimensional formado pelo conjunto de tomografias. Esta malha poligonal, posteriormente, poderá ser inserida na ferramenta de simulação mecânica. Para atingir este objetivo, uma ferramenta de software utilizando uma implementação serial do algoritmo *Marching Cubes* foi testada, gerando resultados pouco satisfatórios.

Quando era requisitado este processamento, o operador da ferramenta de software tinha como retorno um erro de falta de memória, frequentemente depois de muito tempo transcorrido. De forma que, para contornar estes problemas, era necessário diminuir o tamanho da amostra a ser reconstruída ou diminuir a definição do sólido virtual através da eliminação de triângulos e consequente modificação de sua geometria.

Para superar este desafio, a computação heterogênea se apresenta como uma um mecanismo de grande potencial, agregando ao processo uma grande capacidade de processamento, de forma a não apenas permitir reconstruções por meio do melhor uso de hardware disponível, como também realizar as tarefas em um intervalo de tempo mais curto.

Mostra-se de suma importância para o sucesso do projeto em que este trabalho está inserido, portanto, a diminuição do tempo de processamento sem comprometer a definição da imagem, mantendo o máximo de polígonos possíveis no modelo tridimensional gerado. Para tanto, foi implementada uma versão maciçamente paralela do algoritmo *Marching Cubes*, utilizando a GPU (*Graphics Processing Unit*) e que pudesse ser executada a partir de um aplicativo utilizando a plataforma Microsoft .NET, garantindo a compatibilidade com software já utilizado no projeto.

1.3. ORGANIZAÇÃO

Esta dissertação está organizada nos seguintes capítulos:

Capítulo 2: Base teórica e revisão da literatura científica, onde está apresentada uma revisão dos conceitos utilizados no trabalho, bem como o levantamento da bibliografia consultada para a fundamentação teórica.

Capítulo 3: Materiais e métodos científiccos, que apresenta um detalhamento dos equipamentos e das técnicas utilizadas para o desenvolvimento do projeto, bem como os processos técnicos e tratamentos a que foram submetidos os materiais a fim de se obter o produto final.

Capítulo 4: Análise dos resultados, onde foi conduzida uma análise sobre os procedimentos e dados obtidos, bem como as dificuldades encontradas no desenvolvimento do trabalho e as medidas adotadas para superá-las.

Capítulo 5: Discussão e resultados, onde foi recapitulada a motivação para o desenvolvimento do trabalho, com uma posterior análise entre causas e efeitos, os méritos e desvantagens da técnica aplicada, bem como sugestões para trabalhos futuros no sentido de otimizar o que já foi feito ou de adicionar novas funcionalidades de interesse.

Por fim, Capítulo 6: Apresenta as conclusões, considerações finais e síntese do trabalho desenvolvido, bem como sugestões de trabalhos futuros.

2. FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são discutidos, os elementos teóricos utilizados no desenvolvimento do trabalho, bem como os conceitos aplicados e as principais contribuições para sua evolução. Este capítulo não tem a intenção de exaurir os temas, mas levantar as principais bases teóricas em que o trabalho desenvolvido se fundamenta.

Inicialmente, é realizada uma análise dos conceitos de arquitetura de computadores utilizados para o desenvolvimento do trabalho. Em seguida, são apresentados os conceitos de desenvolvimento de aplicativos de software para a arquitetura selecionada. Finalmente, é apresentado o algoritmo a ser utilizado para a extração da isossuperfície de interesse, que posteriormente será utilizada para a simulação virtual do comportamento do corpo de prova.

2.1. GPGPU — GENERAL PURPOSE COMPUTATION ON GRAPHICS PROCESSING UNIT

A arquitetura dos microprocessadores modernos se baseia nos conceitos desenvolvidos por John von Neumann (REED, 2008, p. 250]. Essa arquitetura se baseia na divisão do computador em três componentes principais: 1) a memória; 2) a unidade de processamento central; 3) os dispositivos de entrada e saída. A memória é a responsável pelo armazenamento das instruções e dados, que são alimentados para a unidade de processamento central e os resultados são transmitidos para os dispositivos de entrada e saída, que acumulam as funções de trazer os dados para processamento e de mostrar os resultados para o usuário.

A unidade de processamento central funciona em três etapas: 1) a busca; 2) a decodificação; 3) a execução de instruções [9].

Esses conceitos remontam ao final da década de 40, mas verifica-se que a maior parte dos dispositivos microprocessados são nada mais que a mera personificação do modelo descrito (NVIDIA, 2004), salvo a presença de algumas poucas, mas não desprezíveis, modificações (HENESSY e PATTERSON, 2007, p. 196).

Nas últimas décadas, a frequência de operação dos microprocessadores com um

único núcleo se aproxima do limite físico da matéria em vários aspectos (HENESSY e PATTERSON, 2007, p. 174), sendo alguns deles consumo de energia e capacidade de miniaturização dos componentes. A manufatura de processadores com vários núcleos, como os processadores i3 (INTEL), i5 (INTEL) ou i7 (INTEL) da Intel, vem pavimentar o caminho para a contínua evolução dos microprocessadores.

Para que as operações sejam executadas de forma correta, os microprocessadores necessitam de um espaço de memória em que possam ser armazenados os operadores e os operandos das instruções sendo processadas. Esse espaço de memória é denominado cache, e localiza-se dentro do componente eletrônico que armazena os núcleos (HENESSY e PATTERSON, 2007).

Essa memória corresponde ao mais alto nível da hierarquia de memória. As características principais deste tipo de memória é o pequeno tamanho, em relação às demais memórias da hierarquia, bem como a alta velocidade em comparação com as demais memórias da hierarquia (HENESSY e PATTERSON, 2007).

A CPU é composta, tipicamente, por quatro, oito, dezesseis ou trinta e dois núcleos. Para a arquitetura Intel utilizada nos processadores i3, i5 ou i7 verifica-se a existência de três níveis de memória cache, conforme ilustrado na Figura 1. A memória cache total é de 8 megabytes, sendo que a memória L3 é dividida entre os quatro núcleos e cada um tem memórias L1 e L2 próprias (COOK, 2013).

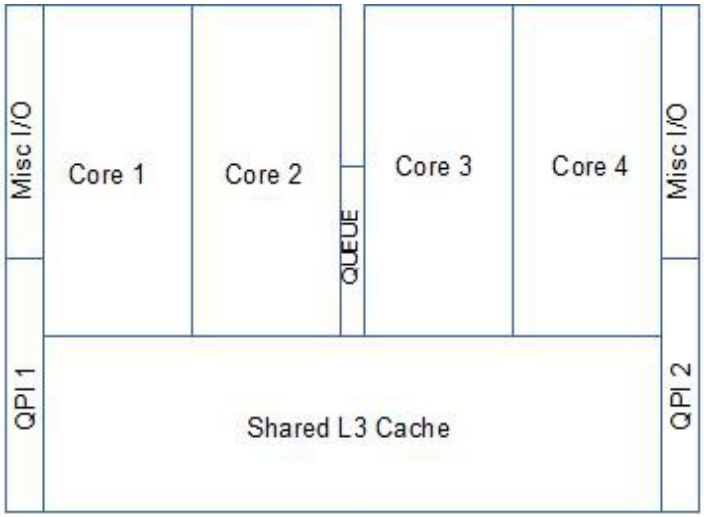


Figura 1: Divisão dos circuitos internos do microprocessador Intel i7.
Reproduzido de (KIRK e HWU, 2011).

Os exemplos de microprocessadores citados, conforme se pode observar na Figura 1, possuem poucos núcleos, realizando simultaneamente poucas tarefas. Para um

maior paralelismo, e um aumento de poder de processamento do hardware, no dia 31 de agosto de 1999 a NVIDIA lançou a GPU, definida da seguinte maneira (NVIDIA, GeForce256):

> “um processador de chip único com mecanismos integrados para a realizar transformações geométricas, renderização, iluminação e disposição e corte de triângulos, capaz de processar, no mínimo, 10 milhões de polígonos por segundo.” (NVIDIA, GeForce256)

Inicialmente, a GPU foi criada para aumentar a qualidade de gráficos tridimensionais em operações como a renderização de mundos e personagens virtuais. A primeira placa de vídeo lançada com uma GPU foi a GeForce 256. Este equipamento tinha uma memória de até 128 megabytes e capacidade de processamento nominal de 15 milhões de triângulos ou 280 milhões de *pixels* por segundo.

A arquitetura mais atual da NVIDIA é denominada Maxwell, com foco no aumento de capacidade de processamento e de memória com uma maior eficiência energética. Entretanto, para o desenvolvimento deste trabalho foi utilizada a arquitetura Kepler, descrita a segiur.

A arquitetura da GPU denominada Kepler pela NVIDIA baseia-se no núcleo SMX, ou Streaming Multiprocessor (NVIDIA). Este é composto por 192 unidades de processamentos. A principal diferença entre os modelos de GPUs utilizando esta arquitetura, entretanto, é a quantidade de núcleos SMX por processador, modelos mais sofisticados possuem mais núcleos que modelos mais simples da mesma família de placas.

A diferença física entre a GPU e a CPU consiste na organização dos núcleos e da memória. A GPU conta com mais núcleos que a CPU. A memória cache, por sua vez, é menor na GPU que na CPU. No campo conceitual, a GPU foi criada para ser um microprocessador que trata *streams* de dados – fluxos de dados que são consumidos após o processamento, sem armazenamento. A CPU, por outro lado, foi criada para realizar o processamento de dados extraídos da memória e armazenar os resultados.

A GPU, construída com a arquitetura Kepler, ilustrada na Figura 2, é composta por milhares de unidades de processamento (mostradas em verde) divididas em núcleos SMX

com 64KB de memória cada, utilizados pelos núcleos da GPU, que compartilham uma memória cache L2 de 1,5MB (em azul claro ambas as memórias) (NVIDIA).

O poder de processamento das primeiras GPUs (Graphics Processing Unit) atraiu a atenção de pesquisadores, que começaram a explorar o poder de processamento das placas gráficas para resolver problemas genéricos utilizando linguagem de shaders, pequenos códigos que rodam na GPU para calcular diferentes efeitos gráficos (COOK, 2013).

Shader é o nome atribuído a códigos que realizam operações gráficas em tempo real. Estão presentes em APIs de computação gráfica, como o OpenGL, e são utilizadas para a renderização de imagens, ao invés de modificar o fluxo de execução ou lógica da aplicação (WRIGHT, HAEMEL, et al., 2011).

Para que o processamento pudesse ser realizado na GPU, os dados deveriam ser enviados para a placa de vídeo através destas linguagens de *shader* e processadas como sendo dados de renderização.

Figura 2: Arquitetura Kepler (NVIDIA).

Esta aplicação das placas gráficas resultou na criação de vários projetos, mas nenhum conseguiu se estabelecer como padrão (COOK, 2013). Para atender essa demanda, desde 2006 a NVIDIA desenvolve a plataforma de computação paralela CUDA (Compute Unified Device Architecture) com um modelo de programação associado (NVIDIA, What Is CUDA). Essa solução vem ao encontro da necessidade de utilizar-se todo o poder da GPU em aplicativos das mais variadas áreas.

Em 2004, a NVIDIA publicou a primeira versão do *GPU Programming Guide* [9], mostrando como melhor utilizar a GPU para paralelizar ferramentas de *software* e diminuir o tempo de execução de algoritmos de computação gráfica. Porém, antes disso, em 2003, o poder de processamento da GPU ultrapassou o da CPU, abrindo campo para a computação paralela heterogênea. O gráfico mostrado na Figura 3 mostra a comparação da evolução do desempenho da CPU e da GPU.

A computação paralela heterogênea leva este nome por ser executado em dois contextos diferentes, a CPU e a GPU (COOK, 2013), sendo que a primeira é denominada *host* (o equipamento hospedeiro) e a segunda *device* (o equipamento subordinado).

O código executado no *host* é, tradicionalmente, serial, enquanto a parte executada no *device* é a parte paralelizada do código, que é executada sem a participação da CPU ou da memória do computador. Na arquitetura Kepler, inclusive, a própria GPU pode iniciar novas *threads* – contextos de processamento paralelos e independentes – e lhes passar os dados para execução, diminuindo ainda mais a participação da CPU na execução de códigos paralelos.

Para melhor compreender a importância deste fato, deve-se analisar a lei de Amdahl (GEBALI, 2011), que enuncia o *speedup*, ou seja, o aumento de velocidade de execução de um determinado código paralelizado em relação à sua implementação serial, com base na parcela paralelizável do código e o número de processadores em que executa. A equação que rege esta lei é ilustrada na Equação 1:

$$S(N) = \frac{1}{(1-f) + \frac{f}{N}} \tag{1}$$

sendo:

S(N) – o *speedup* do código paralelo em relação ao código serial;

f – a parte paralelizável do código. A parte serial do código é, portanto, 1-f. Representado por um valor entre 0 e 1 ;

N – número de processadores executando o código em paralelo.

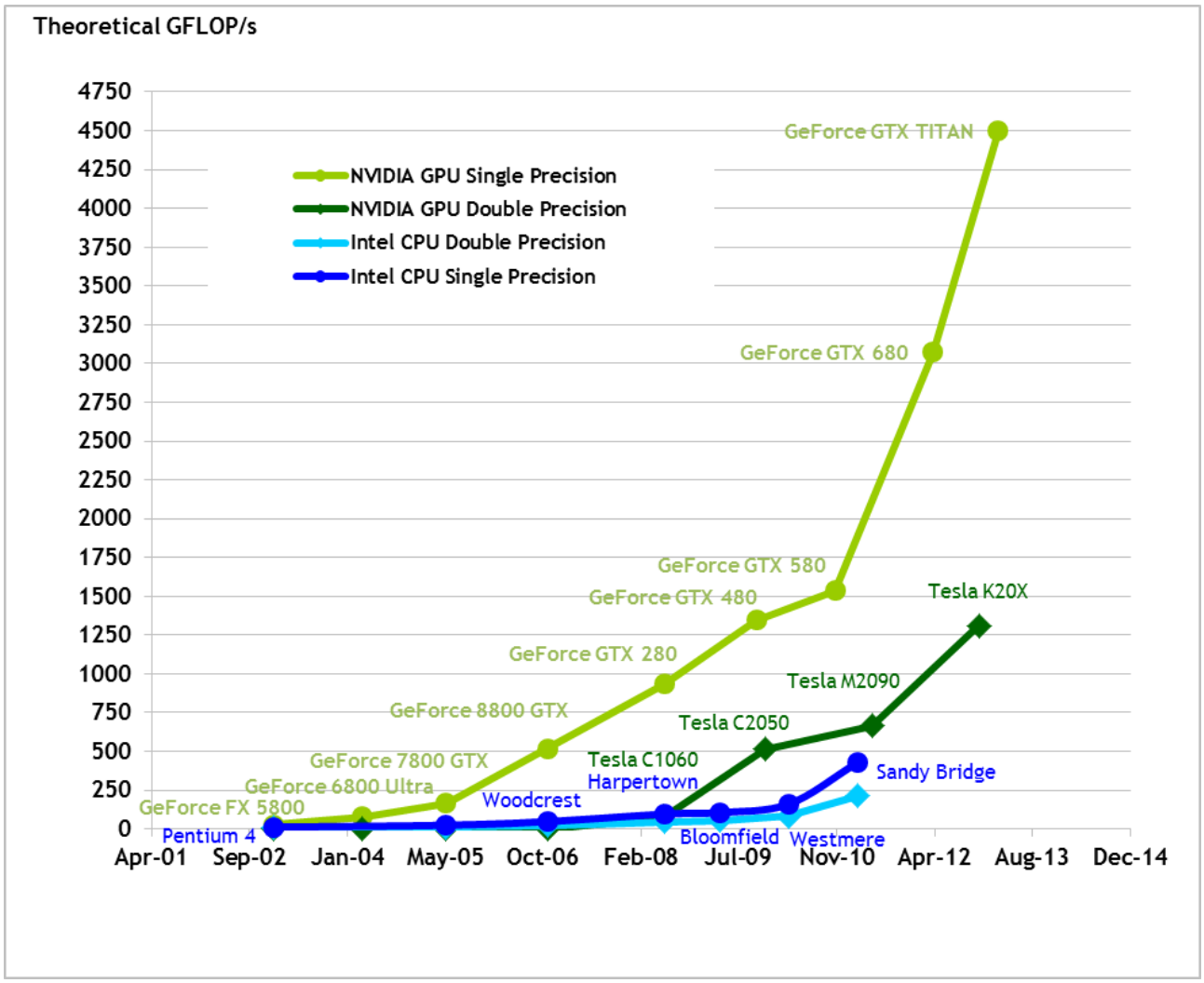


Figura 3: Operações de pontos flutuantes, comparativo entre CPU e GPU (NVIDIA, WhatIsCUDA).

A Figura 4 mostra a evolução do *speedup* conforme aumenta o número de microprocessadores em que o código serial é executado segundo a Lei de Amdahl. Nota-se que quanto maior a parcela paralelizável do código, maior o *speedup*. Se o código for completamente paralelo, o ganho de velocidade será igual à quantidade de processadores, resultando em S(N) = N;

Na prática, entretanto, sempre haverá uma parcela serial (leitura de arquivos, alocação e acesso à memória, pós-processamento), o que significa que 1-f sempre será

diferente de 0. Este valor é o que limita a diminuição do tempo de processamento do algoritmo paralelizado.

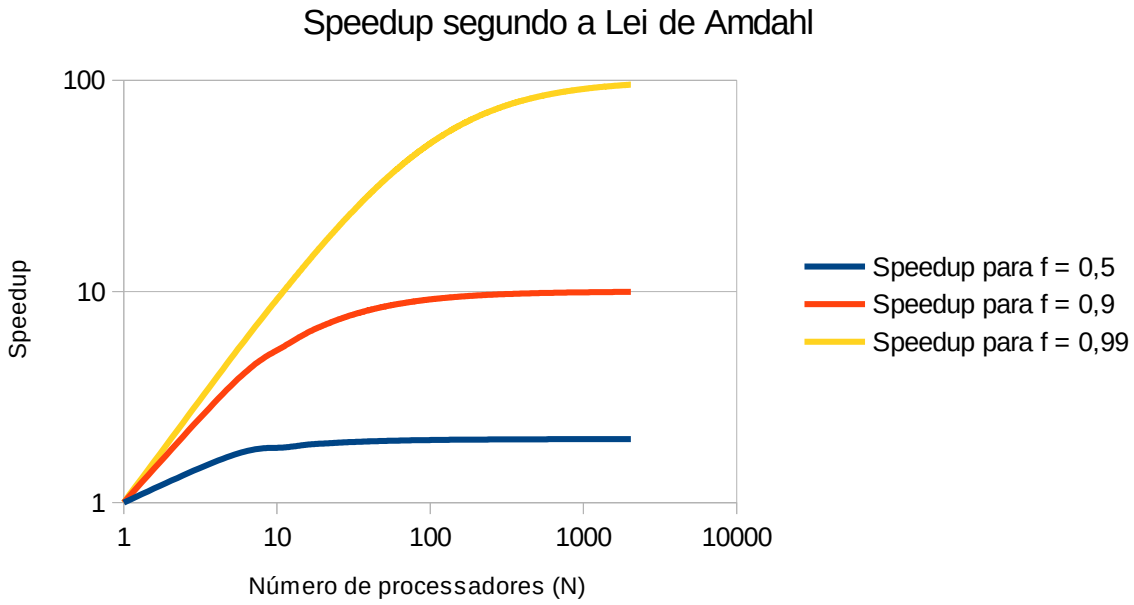


Figura 4: Gráfico mostrando a evolução do speedup de acordo com o número de processadores segundo a Lei de Amdahl. Reproduzido de (GEBALI, 2011)

De fato, a verdadeira conclusão que se pode tirar da Lei de Amdahl é que o ganho de velocidade do código paralelo é limitado pela sua parcela serial. Quanto mais código serial, maior a limitação de ganho de velocidade.

Outra análise que se pode fazer sobre a paralelização de algoritmos, é através da Lei de Gustafson-Barsis (GEBALI, 2011), enunciada na Equação 2:

$$S(N)=1+(N-1)f \tag{2}$$

sendo:

S(N) – o *speedup* do código paralelo em relação ao código serial;

f – a parte paralelizável do código, logo, a parte serial do código é 1-f. Representado por um valor entre 0 e 1 ;

N – número de processadores executando o código em paralelo.

Neste caso, é possível notar que desde que f(N-1) seja maior que a unidade, ocorre o *speedup* do código. O gráfico que demonstra a evolução da velocidade de

processamento em relação à quantidade de processadores executando as instruções em paralelo é mostrado na Figura 5.

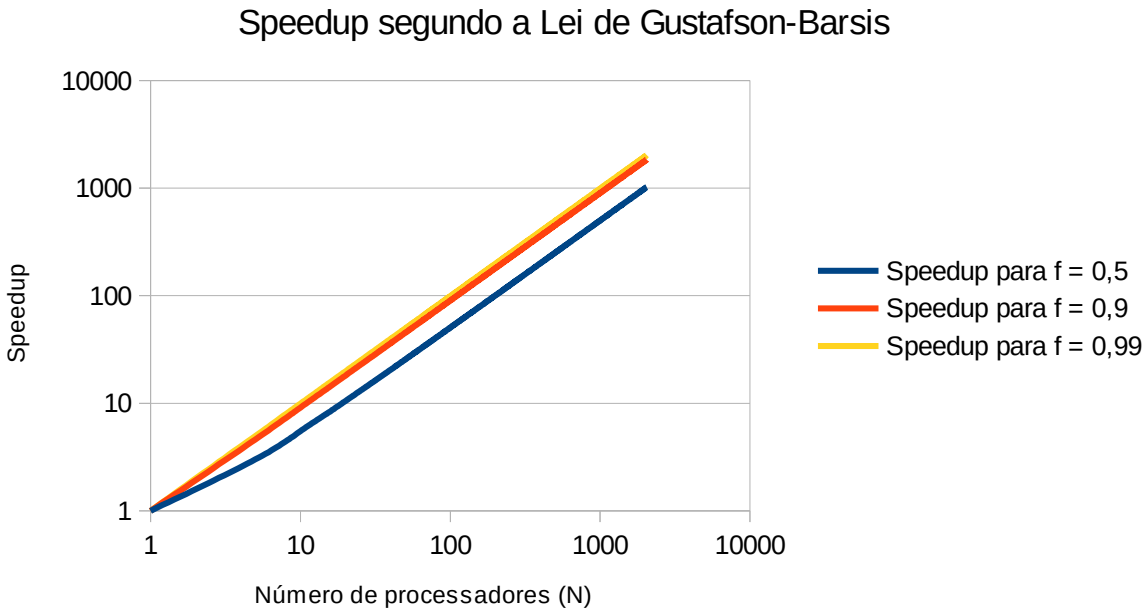


Figura 5: Gráfico mostrando a evolução do speedup de acordo com o número de processadores segundo a Lei de Gustafson-Barsis. Reproduzido de (GEBALI, 2011)

Esta formulação demonstra o poder que a quantidade de processadores tem sobre o tempo de execução de um código que tenha pelo menos uma parcela paralelizada, mostrando mais a influência do equipamento sobre o tempo de execução do que a influência da paralelização do código em si.

As duas formulações apresentadas, portanto, são complementares entre si. A Lei de Amdahl fala sobre a influência das parcelas paralela e serial sobre a redução do tempo de processamento do código. A Lei de Gustafson-Barsis enuncia a influência do número de processadores sobre a velocidade de pocessamento.

A GPU, como já foi verificado, conta com uma grande quantidade de processadores, tornando-a passível de grandes ganhos de velocidade, sendo necessária uma plataforma de programação que permita que o código seja desenvolvido de forma massivamente paralela também. Para o desenvolvimento de código que tire proveito desta característica, utiliza-se o CUDA para o desenvolvimento de aplicativos.

2.2. CUDA

O *Compute Unified Device Architecture* (CUDA) é uma arquitetura de computação paralela desenvolvida pela NVIDIA (FARBER, 2011) que permite a programadores utilizar a GPU para resolver problemas genéricos, não exclusivos de computação gráfica e renderização de imagens virtuais.

A arquitetura CUDA consiste na divisão do ambiente de desenvolvimento em duas unidades lógicas, o *host* e o *device*. O host corresponde à máquina em que a GPU está instalada e que responde pela parte do código que executa na CPU. Em comparação, o *device* é a placa onde está instalada a GPU propriamente dita, que se encarrega da parte massivamente paralela do código elaborado (INTEL, developers).

O CUDA possui um modelo de programação que pode ser visto como um modelo fragmentado, uma vez que o programador tem que pensar sobre a parcela do código que será executada na CPU e na GPU, bem como das transferências de memória entre esses módulos da aplicação (MLAIK, SHARIF et al. 2012).

Esse modelo de programação obriga o desenvolvedor a explicitar tanto o código *host* quanto o código *device*, cada um com seus espaços de memória específicos, ou seja, um dado que esteja na memória do *host* não pode ser acessado pelo *device* e vice-versa.

Esse modelo obriga o desenvolvedor a fazer a carga dos dados no espaço de memória em que será realizado o processamento. Para o *host*, o procedimento é o mesmo que se usaria para carregar qualquer tipo de dado, seja por meio da leitura de um arquivo ou do resultado de cálculos. O *device*, entretanto, deve ser carregado através do *host*, ou seja, os dados para o processamento massivamente paralelo devem, inicialmente, ser carregados pelo código serial encarregado de alocar a memória no *device* e de forma análoga, os dados devem ser enviados para a memória alocada.

O processo de carga de dados para processamento no *Device* está ilustrado na Figura 6. Destaca-se a importância do *Host* no procedimento, verificando-se que o caminho para a carga de dados para processamento paralelo passa, obrigatoriamente, por esse componente. Aqui, cabe ressaltar que isso é consequência natural ao *Device* corresponder a um dispositivo interno ao *Host*.

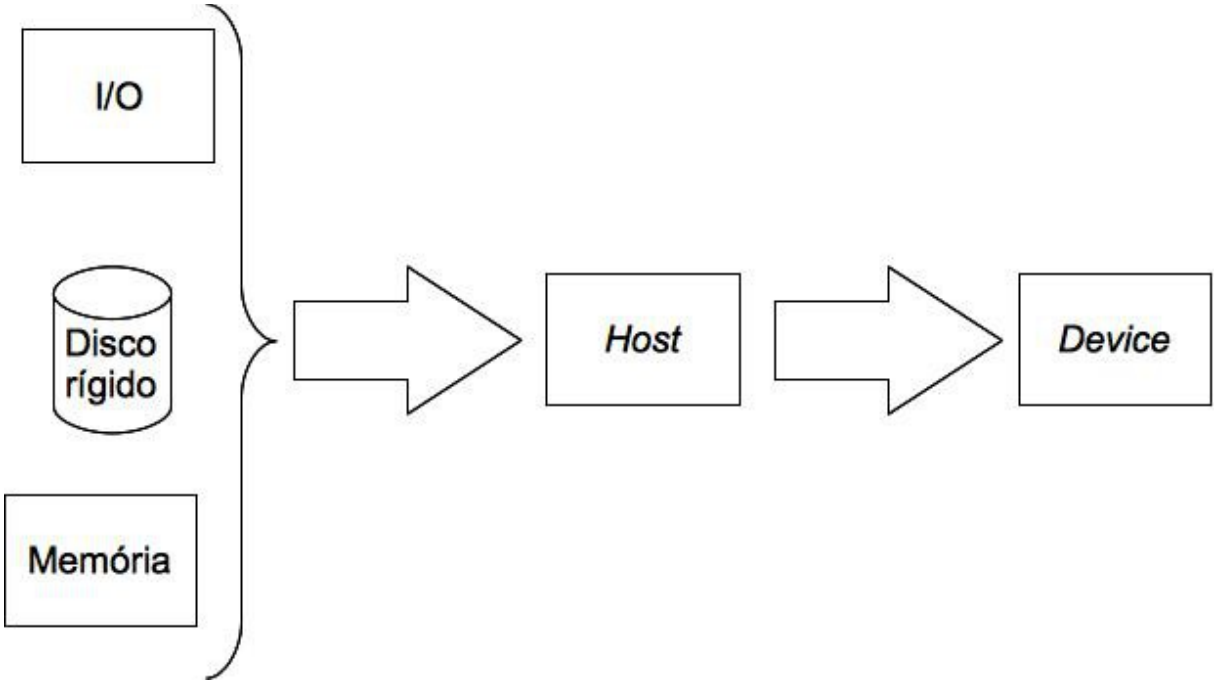


Figura 6: Fluxo de dados para o *device*. Fonte: o autor.

A forma de desenvolvimento do código também é diversa do modelo tradicionalmente aplicado, uma vez que ao se usar o CUDA deve-se designar quais parcelas do código devem ser executadas em paralelo na GPU. Essas parcelas do código devem, inclusive, ser colocadas em um outro arquivo, que será interpretado pelo compilador como uma função que será invocada pelo *Host*, mas deve ser executada no *Device*, com os dados disponíveis em seu espaço de memória.

2.3. MARCHING CUBES

O algoritmo *Marching Cubes* foi descrito pela primeira vez em (LORENSEN e CLINE, 1987), por dois pesquisadores da empresa *General Electric*, William E. Lorensen e Harvey E. Cline. Nesse artigo o algoritmo é apresentado como uma forma de reconstruir isossuperfícies para aplicações médicas a partir de imagens tomográficas ou de ressonância magnética.

Inicialmente, o algoritmo organiza em paralelo todas as fatias tomográficas, alinhadas ao longo do eixo Z, sendo que o plano formado pelos eixos X e Y é paralelo às imagens, criando um espaço tridimensional a partir das fatias bidimensionais. Essa

organização das imagens está representada na Figura 7. Após o alinhamento das imagens tomográficas, são criadas divisões correspondentes a células cúbicas a partir de conjuntos de oito *voxels* adjacentes. Feita esta divisão, tem-se um conjunto de células em que cada vértice possui, além das coordenadas X, Y e Z, um valor numérico correspondente à densidade do material representado.

Para a reconstrução do sólido, define-se um valor numérico que representa a densidade do material de interesse e verificam-se sequencialmente as células, uma a uma comparando o valor de densidade dos *voxels* com o valor de referência. Em seguida compara-se a célula com um dos casos presentes na Figura 8. Por exemplo, o caso 0 em que todos os vértices estão abaixo (ou acima) do valor limite e neste caso, portanto, não há superfície definida. Em contraste, no caso 1 um dos vértices está acima (ou abaixo) do valor limite, enquanto os outros estão abaixo (ou acima) do mesmo valor. Deve-se, então, criar uma superfície que isola o vértice dos demais. Assim, cada um dos casos demonstra como se definem as superfícies reconstruídas do sólido. É importante ressaltar que os casos ilustrados abrangem também suas semelhanças, derivadas por meio das operações de rotação e simetria que são em um total de 256, ou 2^8,casos.

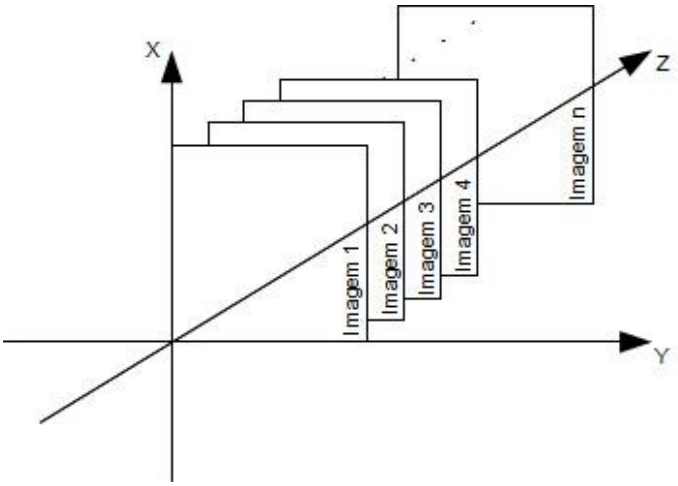


Figura 7: Organização das imagens no algoritmo *Marching Cubes*. Fonte: o autor.

A escolha do caso a ser aplicado se baseia unicamente na posição em que os pontos ultrapassam o valor de referência, mas ainda é previsto um aprimoramento para a superfície por meio de uma função linear aplicada ao vetor normal dos polígonos que formam a superfície reconstruída.

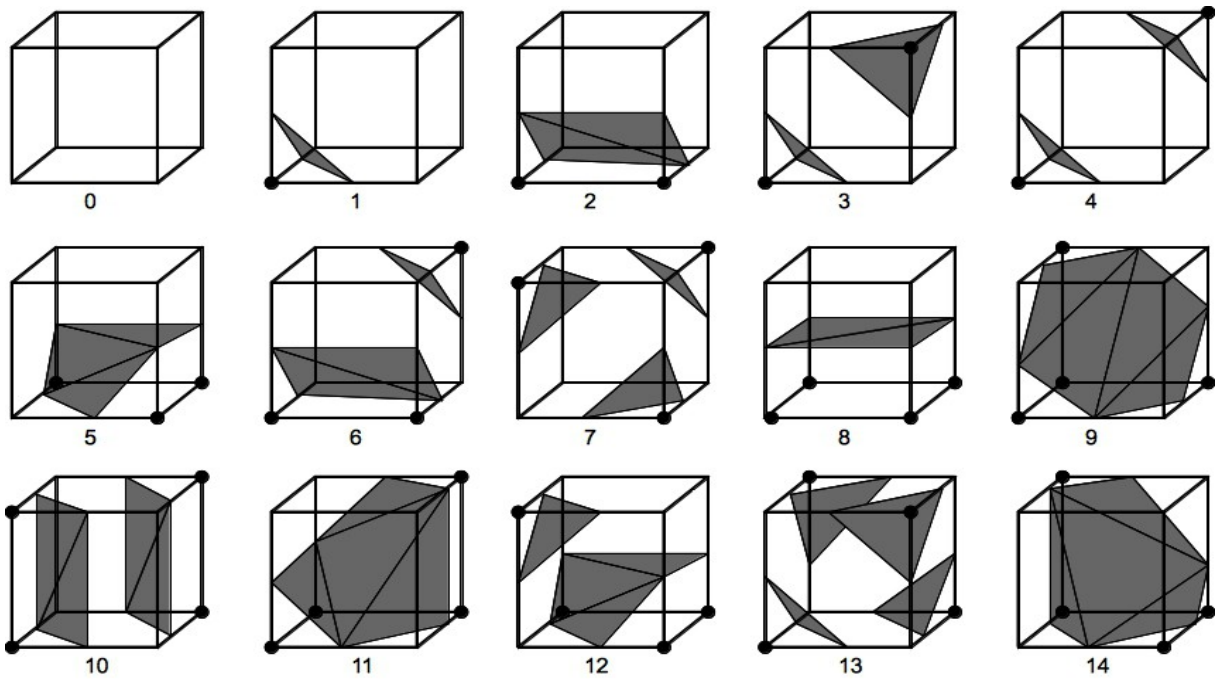


Figura 8: Casos em que uma isossuperfície pode interceptar as células segundo o algoritmo *Marching Cubes*. Reproduzido de (LORENSEN e CLINE, 1987).

Os vetores normais à superfície, necessários para os cálculos de iluminação e tonalização (*shading*), são obtidos a partir pelo gradiente apresentado nas equações 1, 2 e 3 (SUH, 2014). Cada uma das equações reproduzidas corresponde às componentes X, Y e Z do vetor normal – equações 1, 2 e 3, respectivamente.

Nota-se que as equações nada mais são que a interpolação linear das coordenadas dos vértices em relação à diferença de densidade entre os *voxels*.

$$G_x(i,j,k) = \frac{D(i+1,j,k) - D(i-1,j,k)}{\Delta x} \quad (1)$$

$$G_y(i,j,k) = \frac{D(i,j+1,k) - D(i,j-1,k)}{\Delta y} \quad (2)$$

$$G_z(i,j,k) = \frac{D(i,j,k+1) - D(i,j,k-1)}{\Delta z} \quad (3)$$

sendo:

G_x , G_y , G_z = Componente do gradiente no eixo X, Y e Z, respectivamente;

D(i, j, k) = Densidade no *pixel* (i, j) na fatia k;

Δx, Δy e Δz = Os comprimentos das arestas do cubo.

Após realizar os cálculos descritos nas equações 1, 2 e 3, pode-se utilizar o vetor obtido para a tonalização do objeto reconstruído através de modelos de iluminação como o de Phong (FOLEY, DAM et al. 1997) ou de Gouraud (FOLEY, DAM et al. 1997).

Para o modelo de iluminação de Phong, utiliza-se o vetor normal para calcular não apenas a reflexão especular da iluminação como também para o cálculo do ângulo de reflexão em relação ao ângulo de incidência da luz e o ângulo da posição do observador em relação à superfície observada. O método de Gouraud necessita dos vetores normal sobre todos os vértices da malha poligonal e utiliza estas informações para determinar a intensidade de saturação da cor dos vértices e aplicar um modelo de iluminação, aplicando o shading para os sólidos em questão.

Ilustrando o funcionamento do algoritmo Marching Cubes, inicialmente realiza-se o alinhamento das imagens, conforme mostra a Figura 9. Esta etapa é de pré-processamento, sendo realizada no momento da leitura dos dados.

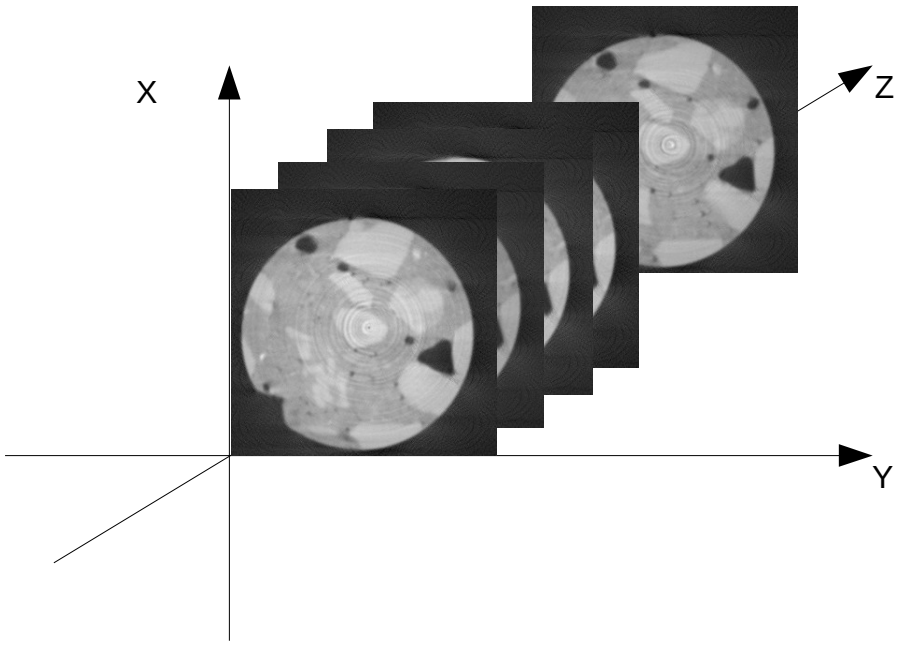


Figura 9: Alinhamento das fatias tomográficas para o processamento pelo algoritmo Marching Cubes. Fonte: o autor.

Em seguida, é realizada a divisão do volume em células de formato cúbico a partir de 8 voxels adjacentes, conforme ilustrado na Figura 10.

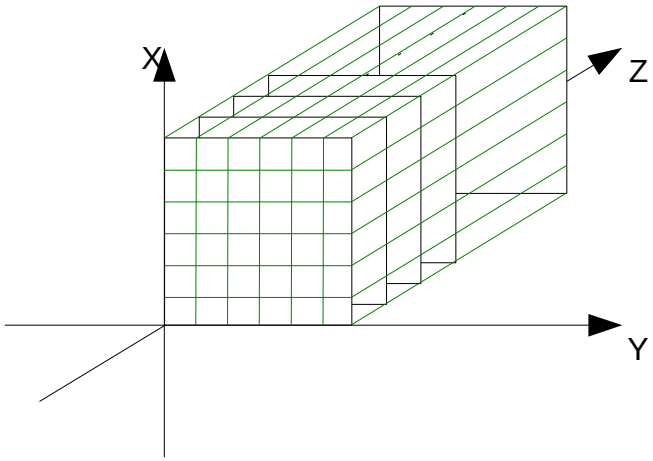


Figura 10: Divisão das imagens em células cúbicas, utilizando oito voxels adjacentes. Fonte: o autor.

Nesta etapa da execução do algoritmo, os dados estão formatados e prontos para que a execução se inicie. Por razões didáticas, será apresentada aqui a forma iterativa do algoritmo, não a paralela. Após a explanação, serão realizadas as considerações para a paralelização do algoritmo. Esta simplificação momentânea é realizada tanto por ser mais fácil de compreender a execução iterativa do algoritmo, quanto pelo fato de o conceito da paralelização ser de fácil compreensão.

Para o processamento propriamente dito, escolhe-se uma célula inicial para ser a primeira a ser processada, conforme mostrado na Figura 11.

A malha poligonal é formada mediante o processamento das relações entre os valores dos vértices da célula. A isossuperfície existe entre dois vértices adjacentes quando o valor de um deles é menor que o da isossuperfície desejada e o do outro é maior ou igual ao da isossuperfície desejada.

A memória destinada a esse processamento é alocada pelo algoritmo original independentemente do fato de a isossuperfície existir ou não dentro daquela célula. Além disso, a memória alocada é suficiente para comportar o caso de maior consumo.

O processo de identificação da isossuperfície dentro da célula é ilustrado na Figura Figura 12.

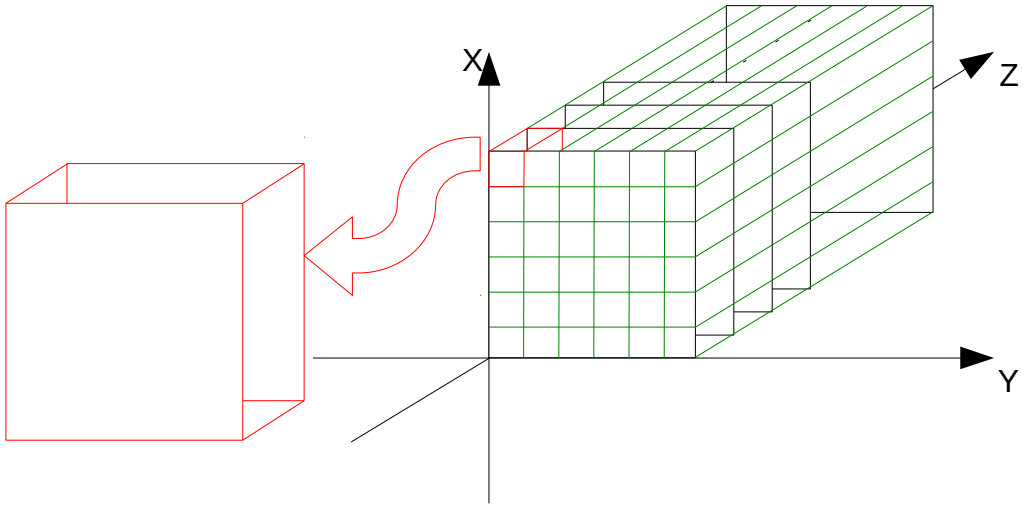


Figura 11: Passo inicial da análise das células cúbicas, escolhe-se uma célula para processar os valores escalares correspondentes aos vértices. Fonte: o autor.

Comparando-se a Figura 12 com os casos do algoritmo, ilustrados na Figura 8, pode-se verificar que é uma rotação do caso 5, sendo necessário, portanto, criar uma parcela para a isossuperfície mostrada na Figura 13.

Após o processamento da célula atual, com o armazenamento dos vértices dos triângulos identificados, o algoritmo avança para a célula seguinte, e assim por diante até que todo o volume tenha sido processado.

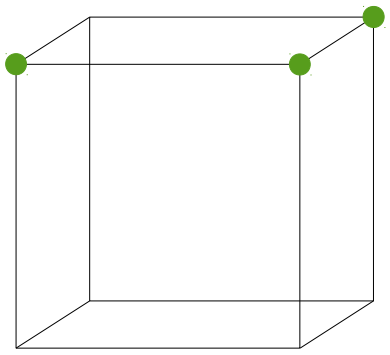


Figura 12: Célula com vértices maiores ou iguais a um isovalor marcados. Fonte: o autor.

Este algoritmo, conforme já mencionado anteriormente, foi desenvolvido para ser uma abordagem dividir e conquistar para solucionar o problema. Esta característica se verifica, uma vez que o processamento de cada célula é independente. Esta característica é que permite a execução em paralelo da técnica.

Para que seja realizada a paralelização, atribui-se uma célula para cada

processador que esteja executando a solução do problema e posteriormente é feita a mera união das soluções das células individuais.

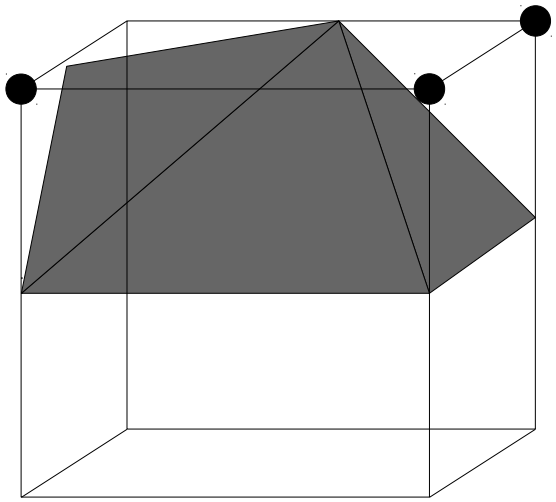


Figura 13: Isossuperfície reconstruída para os vértices marcados na Figura 12. Fonte: o autor.

Como resultado do processamento do algoritmo *Marching Cubes*, tem-se uma malha poligonal. Este processo, descrito em 1987, já foi adaptado de várias formas como se pode verificar em (NIELSEN, 2004), (DYKEN, ZIEGLER et al., 2008) e (ZHANG, NEWMAN, 2003). A possibilidade do algoritmo ser paralelizado já foi explorada em várias ocasiões. Em (MILI, MAHMOUD et al., 2012) foi feita uma análise que comprova que o algoritmo é, de fato, paralelizável.

Além disso, a migração do algoritmo *Marching Cubes* para a GPU e a utilização do processamento paralelo massivo já foi documentada até mesmo pela própria NVIDIA (ZIEGLER, DYKEN, 2010).

2.4. PLATAFORMA .NET

Em 2002 a Microsoft introduziu ao público a plataforma .NET (FREEMAN, 2010) objetivando um modo mais simples e prático para que códigos implementados em linguagens de programação diferentes, tais como C++, Visual Basic e C#, fossem compatíveis no ambiente Windows e componentes de *software* desenvolvidos em uma linguagem pudessem ser utlizados em um sistema desenvolvido em outra linguagem, um

problema bastante comum (FREEMAN, 2010) para o desenvolvimento de aplicativos complexos, com vários módulos desenvolvidos por equipes distintas. Para atender a esta demanda era utilizado o modelo de programação COM, que permitia a interação no mesmo sistema de componentes de *software* desenvolvidos em linguagens diferentes. Porém, a estabilidade desse modelo era muito frágil e a execução das tarefas muito complexa.

Para atingir os objetivos expostos anteriormente, a plataforma é composta por três blocos principais; o *Common Language Runtime* (*CLR*), o *Common Type System* (*CTS*) e o *Common Language Specification* (*CLS*). Estes itens de baixo nível se relacionam de forma a possibilitar o desenvolvedor a trabalhar com uma biblioteca de classes versátil e com a possibilidade de interface com outras linguagens, suportando, inclusive, relacionamentos entre linguagens diversas. Adicionalmente, o *CLR* oferece uma máquina virtual, que corresponde a um ambiente de execução abstrato e ideal (HOGENSON, 2006) para códigos desenvolvidos na linguagem de execução intermediária *Common Intermediate Language* (*CIL*), implementa o *Virtual Execution System* (*VES*) e provê facilidades como um gerenciador automatizado de memória (*garbage collector*), que faz a limpeza automatizada da memória não mais utilizada, por exemplo.

No alto nível, encontram-se mais dois itens de interesse; a *Common Intermediate Language* (*CIL*) e o compilador *Just In Time* (compilador *JIT*). Esses dois itens participam do processo de compilação dos aplicativos desenvolvidos no ambiente .NET. Quando o desenvolvedor inicia o processo de compilação do sistema desenvolvido, o código é compilado para a linguagem intermediária *CIL* e pode ser distribuído para os usuários. Quando o usuário executa pela primeira vez o sistema desenvolvido o compilador *JIT* realiza a compilação do código em *CIL* para o código de máquina.

A compilação intermediária para a *CIL* visa garantir a compatibilidade entre as linguagens do ambiente .NET e a independência de hardware. A primeira característica citada é atingida naturalmente já que todas as linguagens são compiladas para a *CIL* e a integração dos diferentes sistemas é realizada nessa camada. A outra característica é uma consequência natural, uma vez que ao compilar o mesmo código para diferentes plataformas, tem-se um *software* nativo do ambiente alvo, o que significa, por exemplo, o mesmo código pode ser utilizado em um dispositivo portátil ou em um servidor web. Naquele, o foco do código de máquina seria, por exemplo, a eficiência do uso de memória

e bateria, enquanto neste as mesmas restrições não são relevantes.

Aplicativos desenvolvidos neste contexto são executados por uma máquina virtual interna ao *CLR* e são denominados “gerenciados”, ao passo que aplicativos desenvolvidos em ambientes independentes, que não fazem uso dessas funcionalidades, são denominados nativos ou não-gerenciados. Este último tipo de código, após compilado, origina um código executável em linguagem de máquina que pode ser utilizado em conjunto com um outro aplicativo desenvolvido que esteja sendo executado pela *CLR*. Para que seja alcançada esta comunicação entre códigos, a plataforma .NET conta com classes e funções para garantir a interoperabilidade entre códigos. Esta plataforma se denomina P/Invoke e trabalha sobre bibliotecas de vínculo dinâmico (DLLs) (HOGENSON, 2006).

Uma biblioteca de link dinâmico nada mais é que uma função compilada em um arquivo diferente do executável da aplicação e que é ligada ao código em tempo de execução, não de compilação.

2.4.1. INTEROPERABILIDADE NA PLATAFORMA .NET

Apesar de ser uma plataforma pensada para interoperabilidade entre linguagens, ainda é necessário manter a compatibilidade dos sistemas novos com código legado, anterior à criação da plataforma .NET. Para atender a essa demanda, foi criado um sistema de interoperabilidade entre código gerenciado e nativo. Essas classes e funções devem fazer a transformação dos tipos de variáveis entre os contextos, realizando a correspondência entre as variáveis da *CLR* .NET com as variáveis do ambiente não-gerenciado.

Um dos métodos para realizar a interoperabilidade é o *Platform Invoke*, também denominado *P/Invoke*, usado quando o sistema não tem acesso ao código-fonte das funções que são executadas, por exemplo, quando o código invoca uma função armazenada em uma biblioteca de link dinâmico pré-compilada (HOGENSON, 2006). É importante notar que no contexto gerenciado a *CLR* garante a estabilidade do *software.* No não-gerenciado, entretanto, a estabilidade do *software* fica a cargo do desenvolvedor da função externa à *CLR*, sendo esse código passível de falhas.

O conceito básico para a elaboração do *P/Invoke* é que deve ser declarada uma função dentro do contexto da *CLR* que é associada à função implementada pela biblioteca

dinâmica (HOGENSON, 2006). Na prática, quando o desenvolvedor invoca a função gerenciada, são definidas as variáveis e passadas para a função nativa.

A complexidade desse método está no momento em que é necessário fazer a conversão dos valores das variáveis gerenciadas para as variáveis não-gerenciadas, já que os tipos de variáveis do contexto da *CLR* frequentemente possuem metadados que não se encontram nos tipos de variáveis nativas, principalmente quando se trata de tipos não primitivos, como estruturas, *strings* e caracteres (HOGENSON, 2006).

A chamada de função em si é simples e direta. Para que esta tarefa seja realizada, deve-se somente definir o arquivo em que se encontra a biblioteca de vínculo dinâmico utilizando o atributo *DllImport* da CLR. Este atributo é colocado no início do texto do código e nele deve constar o nome do arquivo que se deseja utilizar. Neste ponto, a declaração do nome da função que se deseja utilizar é opcional, somente sendo obrigatório caso exista conflito na assinatura da função – duas funções em diferentes arquivos com a mesma assinatura. Neste caso, junto à declaração do nome do arquivo, deve-se declarar a função que será utilizada e em seguida deve ser declarada a função gerenciada que encapsula a função nativa, com suas variáveis de entrada e tipo de retorno – um tipo inteiro, ponto flutuante, ponteiros ou bytes.

Para o correto funcionamento do sistema, o *P/Invoke* cria automaticamente o código necessário para a chamada da função nativa (HOGENSON, 2006). Esse código se encarrega da mudança de contexto (gerenciado para não-gerenciado) e é denominado *managed entry point*. Naturalmente, tanto a chamada quanto o retorno adicionam uma complexidade ao aplicativo, consumindo um pouco mais de tempo do que se fosse uma chamada para uma função dentro do mesmo contexto.

Para a passagem de parâmetros entre os contextos, é necessário realizar a operação chamada de *Marshalling*, que consiste em fazer a conversão dos tipos de variáveis entre tipos equivalentes, somente mudando o contexto em que elas existem (HOGENSON, 2006). Essa operação é realizada por meio da classe *Marshal*, que possui os métodos apropriados para tratar a conversão de tipos de variáveis entre os contextos.

No momento da mudança de contexto, muito do processamento destinado a realizar a transferência de dados entre o ambiente gerenciado e o nativo consiste de operações de *Marshalling* (HOGENSON, 2006). A transferência de tipos primitivos é quase transparente, sendo necessário apenas observar as equivalências entre tipos

compatíveis, por exemplo, o tipo “short” do C++ nativo é mapeado para o tipo “Int16” da *CLR*, o tipo “long” do C++ é mapeado para o tipo “Int32” da *CLR* (HOGENSON, 2006), e assim por diante.

Para o tratamento de ponteiros do C++ – endereços de memória – é necessário utilizar a classe IntPtr ou UintPtr da *CLR*, que é mapeado para um ponteiro para o tipo void. Este mapeamento foi realizado desta forma para que a classe da *CLR* pudesse conter qualquer tipo de ponteiro e se tornar mais versátil, não precisando cuidado com o tipo de dado passado entre os contextos. Para retirar os dados deste tipo de variável a classe *Marshal* possui os métodos apropriados que podem se encarregar do tratamento da conversão para quase toda forma de tipo da *CLR*.

2.5. FUNDAMENTAÇÃO TEÓRICA RELACIONADA

A reconstrução de isossuperfícies, a simulação não destrutiva de amostras de concreto e a tomografia como ferramenta para a avaliação de estrutras de concreto são problemas estudados por vários autores. Nesta seção serão descritos trabalhos relacionados a estes temas, com uma breve explanação sobre os desafios encontrados e as medidas adotadas pelos autores para resolvê-los.

Em (HANSEN e HINKER, 1992), os autores apresentam uma forma de paralelizar o algoritmo *Marching Cubes* utilizando 64.000 CPUs rodando em paralelo, e mais um número não especificado de núcleos virtualizados.

Em (NEWMAN e YI, 2006), são discutidas formas de modificar o algoritmo básico. Inicialmente, os autores demonstram que um dos casos do algoritmo é redundante, podendo ser eliminado com operações de espelhamento. Em seguida, são discutidas formas de expandir o algoritmo, utilizando células não cúbicas, utilizando o aumento de dimensões do espaço do dataset e verificando os efeitos no tempo de processamento.

Ainda nesse artigo, os autores discutem formas de diminuir o volume de computação, pela eliminação de parcelas do dataset, utilizando octrees, métodos de propagação e *span-based methods*. Um estudo sobre o aumento da complexidade do algoritmo original após a adoção de cada um desses métodos é apresentado, comparando os benefícios obtidos com a inserção de mais cálculos com o ganho obtido.

A paralelização do algoritmo também é discutida, com uma explicação sobre os métodos de balanceamento de carga para processamento, tanto dinâmicos quanto

estáticos. Esses métodos não levam em consideração a utilização do ambiente da GPU para execução, uma vez que apenas a CPU é utilizada nessa análise.

Por fim, os métodos para solução dos problemas de triângulos redundantes, consistência da malha poligonal e ambiguidades são abordados.

Uma simplificação do algoritmo é apresentada em (MONTANI, SCATENI et al., 1994) para datasets discretos, ou seja, que só tenham os valores 1 e 0. Esta simplificação permite que os vértices dos triângulos das isossuperfícies sejam colocados no ponto médio das arestas das células em que o espaço é dividido.

Para a geração da malha poligonal, é utilizada uma pilha. Cada operação da pilha determina o deslocamento para o próximo vértice, baseado na posição do vértice anterior. Como em (NEWMAN e YI, 2006), é demonstrado que um dos casos é redundante e pode ser eliminado utilizando a mesma operação de espelhamento.

Em (CIRNE e PEDRINI, 2013), são apresentados os conceitos do Marching Cubes e como eles se aplicam ao contexto da GPU, com uma discussão sobre o problema de geração de buracos na isossuperfície gerada, apesar de não apresentar solução, paralela ou não para a solução do problema.

Os autores também se depararam com o problema de o dataset utilizado exceder a capacidade de armazenamento do *device.* Para resolver este problema, o conjunto de dados é dividido em partes e cada uma processada separadamente, de forma sequencial.

Em um estudo sobre a linguagem de programação utilizada para o desenvolvimento de código massivamente paralelo para a GPU, (MALIK, LI et al., 2012) comparou o desempenho de algoritmos escritos em quatro linguagens, sendo elas: 1.CUDA, 2.OpenCL, 3.Portland Group Inc. accelerator C99 compiler e 4.MATLAB script.

Após analisar as métricas extraídas das implementações, conclui-se que a linguagem para implementação mais eficiente tanto em tempo de execução como em aproveitamento do dispositivo é o CUDA, podendo esta linguagem ser encarada como a de mais baixo nível para as GPUs fabricadas pela NVIDIA.

A tomografia como forma de analisar corpos de concreto quanto a patologias das estruturas. Em (SUZUKI, MORI et al., 2013), este método foi utilizado para avaliar danos estruturais em barragens após o terremoto de 2011 no Japão. Na análise fez-se a correspondência entre tons de cinza das imagens tomográficas e a resistência mecânica

do material. Neste estudo, a comparação foi possível através da análise de imagens em arquivo e imagens obtidas de novos corpos de prova.

Uma técnica mais simples de reconstrução de malhas poligonais a partir de imagens tomográficas foi utilizada em (PAN e LLOYD, 2014) e em (GARBOCZI, 2002). Nesse método é definido um isovalor. Os voxels são um a um julgados pelo valor de densidade. Caso o valor escalar do voxel ultrapasse o isovalor definido, o voxel é marcado como sendo pertencente a um determinado material.

O modelo virtual reconstruído é gerado a partir da união de todos os *voxels* marcados como o material de interesse. Como resultado do processamento obtém-se um modelo composto de vários cubos, ao invés de faces suavizadas, como o produzido pelo Marching Cubes.

Outro método de análise do corpo de prova é utilizado em (BEHNIA, CHAI et al., 2014 e consiste na aquisição dos dados do corpo de prova através de sistemas de ultrassom. Esse método tem como vantagem a possibilidade de ser utilizado em uma amostra sob carga e medir as tensões internas do corpo de prova com tempo real.

Por esse método, entretanto, não é possível obter informações sobre a geometria do corpo de prova ou da estrutura interna que o compõe. Para realizar a visualização das forças medidas em computador, os autores utilizaram um paralelepípedo com dimensões similares ao sólido real. Nenhum dado sobre a composição interna da amostra foi utilizado.

Outro tipo de avaliação de corpos de prova de concreto foi realizada em (STEIN, PETKOVSKI et al., 2013). Esse trabalho se focou na análise da influência de variações térmicas na estrutura do concreto. Em que foi utilizada a tomografia para a aquisição de imagens e informações sobre a geometria e topologia dos corpos de prova. Como haviam artefatos extras presentes nas imagens foi realizado o tratamento dessas com um filtro anisotrópico tridimensional da mediana. Desta forma, os autores conseguiram diminuir a quantidade de ruído nas imagens e realizar a análise das fatias tomográficas. Com base nestes dados coletados, os autores demonstraram um aumento da quantidade de espaços vazios nas amostras de concreto, correspondente à diminuição de durabilidade do material.

3. MATERIAIS E MÉTODOS CIENTÍFICOS

Neste capítulo é detalhado o desenvolvimento do *software* que utiliza o algoritmo *Marching Cubes*, incluindo a implementação do algoritmo para o ambiente da GPU, suas entradas e os produtos do processamento.

A aplicação desenvolvida consiste em um *software* .NET que utiliza uma biblioteca de link dinâmico elaborada com o CUDA/C executando no *device* e que realiza o processamento paralelo das fatias tomográficas para a reconstrução da malha poligonal que representa o sólido original utilizando o algoritmo *Marching Cubes*.

Para a interface com o usuário foi utilizada a tecnologia .NET. A necessidade de uma interface visual com o usuário se verifica devido à necessidade de se realizar a verificação visual do sólido reconstruído, ficando o usuário responsável por detectar inconsistências e notar falhas no modelo reconstruído.

Na interface elaborada em .NET foi inserido um componente para renderização 3D baseado no VTK, um visualizador de imagens tridimensionais que, ao contrário de ferramentas independentes, pode ser incluído como um módulo de *software* inserido em um outro aplicativo. Esta característica do VTK permite que ele se aloje dentro de outro aplicativo de forma mais simples.

A janela que implementa a interface visual traz informações para o operador sobre as fatias carregadas, permite a reconstrução da malha de polígonos referente ao sólido e permite salvar a geometria gerada para utilização em ferramentas computacionais de simulação de resistência dos materiais.

Para o desenvolvimento da janela foi escolhida linguagem de programação C#, devido à facilidade de integração com o VTK. Um efeito colateral não previsto, entretanto, foi a complexidade adicionada à integração entre o *software* e o módulo desenvolvido em CUDA/C, que teve que ser implementado utilizando os módulos de transferência de dados entre o contexto gerenciado e nativo do aplicativo .NET.

3.1. SISTEMA DE AQUISIÇÃO DE IMAGENS TOMOGRÁFICAS

As imagens tomográficas foram adquiridas no equipamento presente no

Laboratório de Instrumentação para Diagnóstico de Materiais dos Institutos LACTEC.

O sistema de tomografia é composto de um detector digital de raios x direto (*flat panel*) de 16 bits com 40 cm x 40 cm (*pixel* de 200 µm x 200 µm), com capacidade de captura de até 100 imagens por segundo, fabricado por PerkinElmer, 01 fonte de raios X milifoco de 225kVp e 10 mA do fabricante Comet , que opera em cabine blindada, com software e instrumentação para aquisição de imagens desenvolvidos no Lactec.

O esquema do sistema tomográfico utilizado para a aquisição das imagens é ilustrado na Figura 14, com os componentes indicados por números.

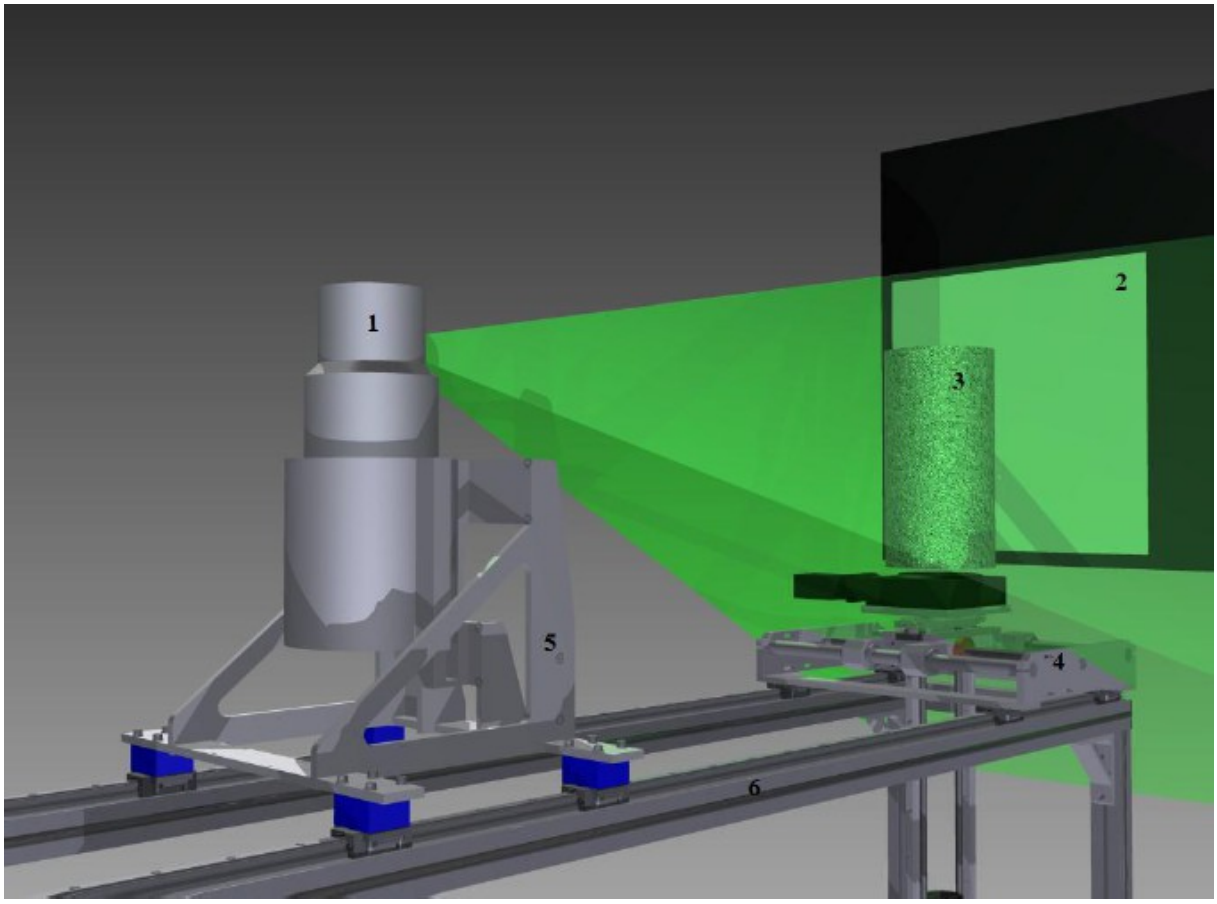


Figura 14: Ilustração esquemática do sistema tomográfico utilizado. Indicados na Figura: 1) tubo de raios X, 2) detector plano de raios-X, 3) amostra de concreto, 4) carro suporte da mesa, 5) carro suporte do tubo de raios-X, 6) estrutura principal. Fonte: equipe do projeto.

3.2. IMAGENS TOMOGRÁFICAS

Para a aquisição das imagens, o sólido a ser analisado é colocado no tomógrafo industrial e as imagens são capturadas pelo operador e armazenadas em arquivos de formato TIFF, em tons de cinza 16 bits. Desta forma, o valor de cada *pixel* pode variar entre 0 e 65535, sendo que o valor 0 corresponde à cor preta e o valor 65535 corresponde ao branco.

Este formato de imagem é utilizado no trabalho devido ao fato de já ser o tipo de arquivo fornecido pelo sistema de processamento de dados do tomógrafo.

O valor do *pixel* é diretamente proporcional à densidade do material obtido na imagem. Ou seja, para o ar o *pixel* assume um valor próximo de 0, enquanto para um material bastante denso o tom de cinza é próximo do máximo. A Figura 15 demonstra uma das fatias tomográficas produzidas pelo tomógrafo.

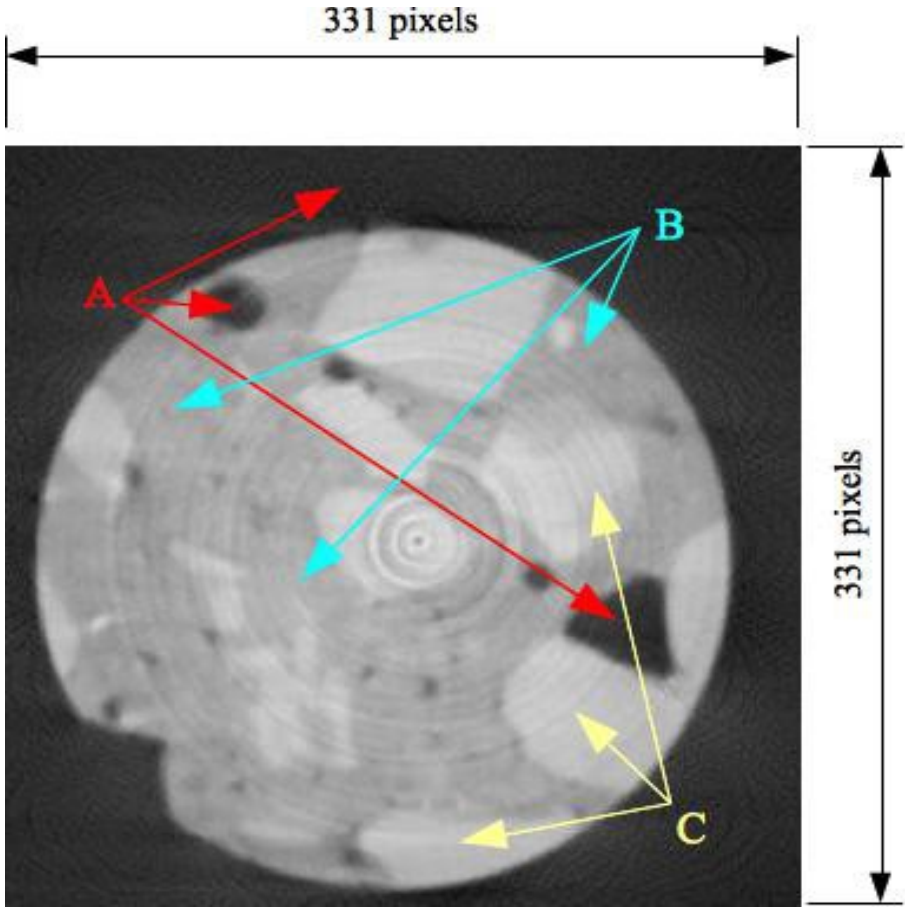


Figura 15: *Exemplo de fatia tomográfica do corpo de concreto.* A letra A indica os espaços vazios, a letra B indica o concreto e a letra C indica o agregado. Fonte: equipe do projeto.

A Figura 15 demonstra claramente os três materiais das amostras de concreto; o ar, marcado com a letra A, a massa cimentícia, letra B, e o agregado, letra C. Nota-se que o ar é consideravelmente mais escuro que os outros dois materiais, criando um limiar de tons de cinza bastante definido que separa o que pode ser chamado de ar, seja na parte externa do corpo de prova ou em cavidades internas do concreto.

Na Figura 16 pode-se observar o histograma correspondente à fatia ilustrada na Figura 15. Verifica-se que há uma grande concentração de *pixels* entre as tonalidades de cinza 8.000 e 20.000.

Outra concentração de *pixels* fica entre 30.000 e 52.000. Esta concentração corresponde aos *pixels* da massa cimentícia e do agregado. Apesar do agregado e da massa cimentícia serem visualmente distintos, a segmentação das imagens automaticamente pelo *software* é não-trivial, uma vez que os tons de cinza dos materiais se sobrepõem, dificultando a catalogação dos materiais distintos apenas mediante um valor limite de tom de cinza, sem a aplicação de filtros.

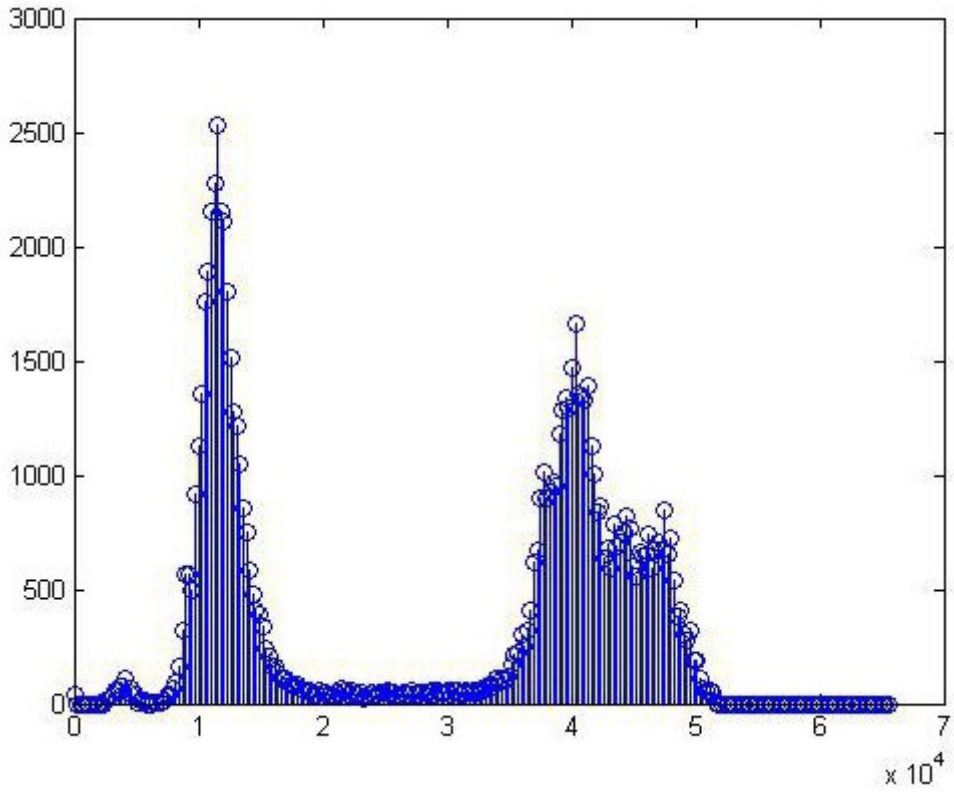


Figura 16: Histograma referente à contagem de pixels por valor de cinza da fatia tomográfica ilustrada na Figura 15, mostrando dois agrupamentos de valores para três materiais. Fonte: o autor.

Desta forma, o primeiro passo para o processamento das fatias tomográficas foi aplicar um filtro às imagens, de forma a facilitar a segmentação dos materiais. Inicialmente foi utilizado o filtro da mediana, e posteriormente foi definida a janela do filtro por meio de um processo iterativo.

Uma vez definido o filtro, este foi aplicado sobre as fatias tomográficas. Estas etapas foram realizadas no MATLAB, que já contém funções específicas e otimizadas para estas operações, apesar de sofrer com o excesso de processamento inerente a linguagens interpretadas.

A característica principal dessas linguagens é o fato de não passarem por um processo de compilação, em que é gerado código executável pelo microprocessador. Por este motivo, o código deve ser interpretado por outro aplicativo e somente então enviado para o *hardware*. Esta operação insere uma complexidade no procedimento, aumentando o tempo de processamento e reduzindo os recursos computacionais disponíveis para a implementação realizada.

3.3. IMPLEMENTAÇÃO DO ALGORITMO *MARCHING CUBES* NA GPU

A função do algoritmo *Marching Cubes* é criar uma isossuperfície tridimensional a partir de uma série de imagens bidimensionais com informações referentes às densidades dos materiais.

Para o projeto em questão, o algoritmo *Marching Cubes* tem como entrada um número variável de fatias tomográficas obtidas a partir do corpo de prova de concreto e alinhadas ao longo do eixo z. O sólido resultante do alinhamento é representado por uma matriz de três dimensões, em que o valor de cada *voxel* corresponde à integral tripla da função densidade ao longo das três direções dentro do volume espacial que o *voxel* delimita. A função densidade é contínua, mas como os dados volumétricos são discretos, cada valor elementar é representado por um valor constante – o valor do voxel.

Esta relação é expressa pela Equação 4:

$$P(x, y, z, V) = \int_{x_i}^{x_f} \int_{y_i}^{y_f} \int_{z_i}^{z_f} V \, dz \, dy \, dx \tag{4}$$

sendo:

x_i, x_f – fronteiras do volume ao longo do eixo coordenado X;

y_i, y_f – fronteiras do volume ao longo do eixo coordenado Y;

z_i, z_f – fronteiras do volume ao longo do eixo coordenado Z;

V – o volume a ser processado;

P – o valor escalar da densidade correspondente ao voxel nas coordenadas definidas.

Conforme já explicado, o algoritmo *Marching Cubes* considera células volumétricas em forma de cubos a partir de voxels adjacentes ao longo dos três eixos a partir de oito *voxels* das imagens alinhadas. Para a implementação, criam-se dois vetores de bits, um para os vértices, outro para as arestas. Cada elemento da geometria leva um código de identificação, na Figura 17 é apresentada a indicação dos vértices e arestas da célula utilizada no algoritmo.

Inicia-se o algoritmo verificando quais vértices são maiores e quais são menores que um valor de densidade arbitrário que serve de referência para a classificação dos materiais e que define quais *voxels* estão dentro do volume de interesse e quais estão fora do volume de interesse. Conforme a quantidade e posição destes *voxels*, utiliza-se um dos casos ilustrados na Figura 8 para a criação dos polígonos da reconstrução da seção da isossuperfície compreendida dentro da célula sendo analisada utilizando uma malha com uma quantidade entre um e quatro triângulos. A união destas parcelas, por fim, gera o sólido reconstruído.

O algoritmo tem em seu núcleo duas tabelas de números inteiros. Um vetor denominado tabela de vértices, com 256 posições e uma matriz de dimensões 256x16. Cada uma das tabelas tem um papel bem definido – uma na determinação das coordenadas dos vértices da isossuperfície e a outra na determinação dos vetores normais a ela.

Para a classificação de cada célula, cria-se um vetor de oito posições, cada posição correspondendo a um vértice, conforme ilustrado pela Tabela 1. Se o vértice estiver dentro do sólido, atribui-se 1 à posição correspondente. Caso contrário, atribui-se o valor 0.

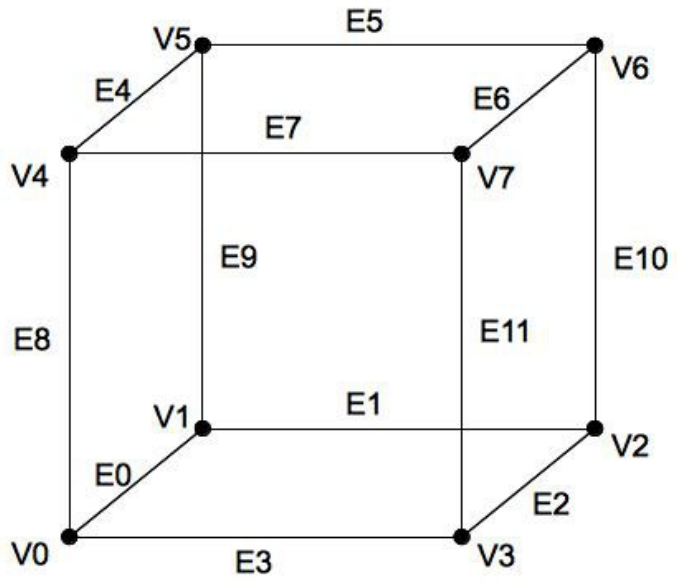


Figura 17: Numeração dos vértices do algoritmo *Marching Cubes*.
Fonte: o autor.

Tabela 1: *Tabela de vértices do algoritmo Marching Cubes.*

Bit	8	7	6	5	4	3	2	1
Vértice	V7	V6	V5	V4	V3	V2	V1	V0

Lê-se o vetor como um número inteiro e utiliza-se este número como índice da tabela de vértices. O valor que a tabela contém na posição em questão é decomposto para um vetor de 12 bits. Cada um dos 12 bits corresponde a uma aresta do cubo, conforme ilustrado na tabela 2. As arestas que estão marcadas com um bit 1 contém um dos vértices dos triângulos do caso em questão.

Tabela 2: *Tabela de triângulos do algoritmo Marching Cubes.*

Bit	12	11	10	9	8	7	6	5	4	3	2	1
Vértice	E12	E11	E10	E9	E8	E7	E6	E5	E4	E3	E2	E1

Para determinar a posição exata do ponto de interseção entre os triângulos e as

arestas do cubo, utiliza-se interpolação linear, conforme descrito na Equação a seguir.

$$P=[(\frac{D_r-D_i}{D_f-D_i})(V_f-V_i)]+V_0 \quad (5)$$

Sendo que:

D_r = Densidade de referência, especificada pelo usuário no início do algoritmo
D_i = Densidade do ponto à esquerda do vértice da isossuperfície
D_f = Densidade do ponto à direita do vértice da isossuperfície
V_i = Coordenadas do ponto à esquerda do vértice da isossuperfície
V_f = Coordenadas do ponto à direita do vértice da isossuperfície

Neste ponto da execução do algoritmo, invoca-se matriz composta de 256 vetores de 16 posições. A tabela contém a sequência em que os vetores dos triângulos da isossuperfície são lidos.

A tabela é utilizada para determinar a direção para a qual o vetor normal aponta. Em outras palavras, determina qual é o lado externo à isossuperfície e auxilia na visualização final do sólido, mediante o uso de algoritmos de shading, conforme já descrito anteriormente.

A implementação deste algoritmo na GPU foi baseada no código encontrado na bibliografia (SUH e KIM, 2014). A implementação é simples, devido à malha de triângulos depender somente do cubo sendo processado, sem a interferência de outros dados.

As tabelas do algoritmo são implementadas diretamente na porção do *software* que roda na GPU, evitando que mais dados, além das fatias tomográficas, tenham que ser transmitidos entre o *host* e o *device*. Outro benefício é o fato que, devido às tabelas já estarem carregadas na memória do *device*, todas as *threads* – uma linha de processamento capaz de dividir recursos do processador com outras linhas de processamento (PARHAMI, 2002) – podem consultar as mesmas instâncias das matrizes, economizando memória.

Para a invocação do método desenvolvido em CUDA, foi criada uma função em C++ que deve ser executada no *host* e que define em tempo de execução quantos núcleos da GPU o processamento utilizará. Outra atribuição desta é fazer a alocação e

desalocação de memória no *device* e realizar a transferência de dados bidirecionalmente entre *device* e *host*.

A importância desta função é que a invocação do código paralelo CUDA não pode ser realizado diretamente pelo aplicativo desenvolvido em C# devido aos diferentes contextos em que os códigos são executados – nativo e gerenciado. Como as ferramentas desenvolvidas no projeto são escritas em C# é necessário prover uma interface compatível com futuros códigos elaborados.

3.4. IMPLEMENTAÇÃO DO *SOFTWARE* .NET

Para manter a compatibilidade com o aplicativo existente, foi utilizada a plataforma Microsoft .NET já apresentada. A linguagem utilizada para esta camada de compatibilidade foi a C#.

Esta escolha se justifica pelo fato de já haver código escrito em C# para o projeto em que este trabalho se insere e o módulo resultante deste trabalho pode ser utilizado para complementar o projeto. Esta é uma limitação imposta para garantir a compatibilidade futura com o projeto corrente.

Conforme já explicado anteriormente, a plataforma .NET utiliza o *CLR* para realizar a execução dos aplicativos de *software* cabíveis. O módulo escrito em CUDA/C, entretanto, não pôde ser implementado para ser executado no mesmo contexto.

Esta distinção fica clara no momento da compilação do código, em que são necessários dois compiladores distintos, cada um responsável por uma parcela do aplicativo desenvolvido – o componente escrito em C# que utiliza a memória gerenciada e a biblioteca de link dinâmico desenvolvida em CUDA/C que é executada no ambiente nativo.

Como os códigos não são compilados juntos, é preciso utilizar o método de comunicação entre os ambientes gerenciado e nativo – *Marshalling* – para realizar a transferência de dados entre os contextos. Esta forma de comunicação demonstrou-se bastante eficiente, porém, não sem complexidade.

O principal desafio que foi necessário superar foi manter a compatibilidade entre os dados utilizados pelos módulos de *software*. Como o ambiente de execução é

diferenciado, a forma de armazenamento das variáveis não necessariamente é similar, gerando erros durante a execução.

Para realizar a tradução entre tipos de dados foram utilizadas funções próprias para a conversão dos tipos de dados e manipulação direta da memória física.

Estes procedimentos precisaram ser adotados tanto para o envio dos dados para processamento tanto quanto para o retorno dos resultados. No caso do retorno houve o agravamento da situação, uma vez que não se verificava a compatibilidade entre os dados que poderiam ser importados para a aplicação sendo executada no sistema gerenciado.

Para o tratamento deste problema foi necessário ler os *bytes* dos dados e por meio de operações lógicas e aritméticas obter os valores corretos a partir do retorno do processamento da GPU.

4. DETALHES DE DESENVOLVIMENTO E RESULTADOS

Para o desenvolvimento do *software* CUDA/C, após pesquisa inicial na bibliografia por procedimentos para desenvolvimento eficiente de código para ser executado pela GPU, foi adotado um modelo de produção em que inicialmente realiza-se a implementação do protótipo do *software* em MATLAB e posteriormente este é transcrito em CUDA/C. Para verificar o funcionamento do módulo elaborado, utiliza-se a interface MEX do MATLAB, explicada em mais detalhes posteriormente.

Para as etapas iniciais de desenvolvimento e depuração do código foi utilizada uma aplicação de exemplo encontrada na bibliografia (SUH, 2014) e posteriormente foi extrapolada a aplicação para que fosse feito o processamento das imagens tomográficas relevantes ao projeto. Este procedimento foi necessário uma vez que a aplicação exemplo tratava somente volumes muito pequenos de dados, tornando sua aplicação muito limitada para este trabalho.

Para o desenvolvimento do protótipo MATLAB foram utilizadas as ferramentas disponíveis no próprio aplicativo. Para o desenvolvimento e depuração do código em CUDA/C foi utilizado um ambiente misto, utilizando-se o depurador do MATLAB, os depuradores do aplicativo de desenvolvimento Microsoft Visual Studio e o NVIDIA Nsight, específico para a análise de módulos que utilizam a GPU.

4.1. PROTOTIPAÇÃO EM MATLAB

O aplicativo MATLAB inicialmente foi desenvolvido utilizando a linguagem de script da própria ferramenta, conforme ilustrado na bibliografia (SUH e KIM, 2014). Neste ponto, foi realizada a medição de tempo de processamento e memória consumida para um conjunto teste.

Posteriormente, conforme (SUH e KIM, 2014), foi implementada uma função script MATLAB utilizando o recurso de vetorização do MATLAB, diminuindo o tempo de processamento. Novamente, foi realizada a medição do tempo de execução e da memória consumida para a operação com o mesmo conjunto de dados teste, verificando-se o decréscimo do tempo de execução sem aumentar o consumo de memória.

A vetorização no MATLAB, segundo (SUH e KIM, 2014), é a aplicação de uma mesma operação simultaneamente em todos os elementos de dois ou mais vetores e/ou matrizes.

Após esta etapa, foi implementada uma biblioteca dinâmica para MATLAB em CUDA/C utilizando a interface de programação MEX, que possibilita a integração de códigos em C ao MATLAB.

Apesar de o MATLAB permitir o processamento em GPU de forma integrada ao script, a elaboração de *software* utilizando o CUDA/C apresenta melhorias em relação à velocidade, trazendo agilidade ao processo. Esta mudança na implementação também gera uma função que pode ser reaproveitada de forma mais versátil.

Para o processamento em GPU de dados do MATLAB, é necessário enviar as variáveis para o *device* utilizando um comando específico e realizar as operações como estivessem no *host*.

Apesar da comodidade do procedimento, sua utilidade é limitada. O sistema sendo desenvolvido precisa ser integrado na forma de uma biblioteca de link dinâmico ao ambiente .NET gerenciado. Este procedimento é dificultado pelo ambiente de desenvolvimento utilizado no contexto do MATLAB. Outra desvantagem deste método persistir no uso de uma linguagem interpretada, quando o modo mais eficiente de implementar a solução é mediante a linguagem compilada CUDA/C, como discutido na seção 3.2.

Para que este sistema possa ser integrado ao MATLAB, deve ser criada uma camada de compatibilização entre os sistemas. Esta camada é dispensável para as etapas seguintes. Desta forma, a biblioteca dinâmica foi criada de forma que posteriormente os trechos de código pudessem ser retirados da forma mais fácil possível.

Após a validação com os dados de teste, foi processado um conjunto menor de fatias tomográficas e tanto o sólido quanto os vetores de vértices e triângulos criados foram utilizados para testes e validação do resultado do processamento do aplicativo.

Para a visualização do resultado do processamento do *software* do MATLAB foi utilizado seu visualizador padrão. Desta forma, foi possível verificar a isossuperfície que resultou do processamento das fatias tomográficas dado um isovalor arbitrário de forma simples.

4.2. DESENVOLVIMENTO DO *SOFTWARE* C#

Esta linguagem de programação foi adotada devido ao fato de já ser utilizada tanto no desenvolvimento de outros aplicativos para o projeto em que se insere este trabalho quanto pela facilidade de integração com o VTK, empregado na visualização da isossuperfície reconstruída pelo algoritmo *Marching Cubes* a partir das fatias tomográficas e posterior exportação do modelo no formato compatível com o *software* de simulação mecânica utilizado pelo projeto.

A linguagem C# foi utilizada também para a criação da interface visual. Um dos requisitos do aplicativo desenvolvido é que possa ser posteriormente utilizado por um operador. Logo, a criação de uma interface amigável ao usuário é importante.

O *software* em C# foi elaborado de forma a realizar um pré-processamento das imagens, já preparando os dados para que a implementação realizada execute as operações necessárias em um conjunto apropriado de dados, sem ter que tratar problemas como imagens fora de ordem ou arquivos em formato incorreto.

Inicialmente, o *software* procura e importa a função *Marching Cubes* contida em sua respectiva biblioteca dinâmica – código compilado armazenado em um arquivo distinto da aplicação e que é referenciada em tempo de processamento – armazenada em um arquivo dll junto com o arquivo executável do *software* C#.

Para início do processamento do aplicativo, deve-se informar o local no computador em que as fatias tomográficas devem ser carregadas.

As imagens são armazenadas em um vetor de números inteiros 16 bits de uma dimensão, sendo que cada posição do vetor compreende um valor correspondente ao nível de cinza no *pixel* da imagem em questão. Este vetor tem que ser então retirado do contexto gerenciado pela *CLR* e enviada ao contexto nativo, ambos descritos no item 2.4.

A função que é executada na GPU pode ser invocada neste ponto. Este é o ponto onde o processamento em si acontece, realizando o tratamento das imagens e construindo a isossuperfície que representa as informações geométricas relevantes retiradas do sólido real. Este processo será tratado em mais detalhes posteriormente, no item 4.3.

A função externa retorna uma referência de memória para números de ponto

flutuantes, mas como esta variável está na memória não gerenciada, é necessário fazer o *Marshalling* para a memória gerenciada. Esta forma foi a escolhida para trabalhar com a comunicação entre contextos de forma a possibilitar ao código em C# processar os valores para a subsequente visualização dos volumes reconstruídos.

A interface visual criada utilizando-se a linguagem C# também é utilizada para a verificação visual do volume gerado. Para que isto possa ser realizado, foram integradas à janela funcionalidades do aplicativo VTK. Esta etapa de validação é importante para verificar se os parâmetros utilizados para a criação da isossuperfície são corretos ou não e, conforme a necessidade, revê-los e recriar o modelo virtual.

Para a criação dos corpos reconstruídos utilizou-se o resultado do processamento da função *Marching Cubes* na GPU, composto por duas estruturas, uma com a topologia do sólido outra com a sua geometria, ou seja, um vetor contém as coordenadas dos pontos do sólido e o outro contém os dados para a definição de triângulos mediante a ligação entre os pontos.

O VTK é utilizado também para a criação de arquivos que possam ser alimentados no *software* de simulação mecânica. Isto é possível já que ambos os aplicativos utilizam o mesmo formato de arquivo, garantindo que o resultado do processamento possa ser posteriormente alimentado na simulação.

Neste ponto, deve ser feito o *Marshalling* dos dados novamente, uma vez que os dados da biblioteca dinâmica devem ser trazidos para o ambiente gerenciado .NET descrito na seção 2.4, desenvolvido em C#. Este processo foi dificultado uma vez que o componente de software responsável pelo *Marshal* necessita de um vetor para números inteiros, codificado na forma de dados IntPtr do C#. Entretanto, este vetor deveria ser lido como um vetor de números de ponto flutuante de três posições.

4.3. CUDA/C

Como o C# é uma linguagem de alto nível e que utiliza a *CLR* para realizar a abstração do *Hardware* (HOGENSON, 2006) as funcionalidades acessíveis somente fora do contexto gerenciado têm uma complexidade agregada à sua implementação.

Para que fosse possível realizar o processamento em paralelo com o CUDA/C, a

implementação da função necessária foi colocada em um arquivo à parte e posteriormente foi ligada ao módulo desenvolvido em C#.

O intuito ao utilizar a função desenvolvida em CUDA/C neste caso foi o de paralelizar o algoritmo *Marching Cubes*, utilizado na reconstrução da geometria dos sólidos a partir das imagens tomográficas.

O CUDA/C é distribuído gratuitamente pelo fabricante, em contraste com o CUDA/Fortran, distribuído por uma entidade independente do fabricante e com um custo financeiro agregado. Dentre as vantagens de utilizar o CUDA/C para desenvolvimento está a integração com o ambiente de desenvolvimento utilizado, tanto para a compilação do código como para sua depuração.

Para a depuração foi utilizado o NSIGHT, também da NVIDIA, integrado ao *software* de desenvolvimento.

Outro recurso do CUDA/C utilizado foi a possibilidade de dividir o processamento entre mais unidades, aproveitando equipamentos que possuem mais recursos disponíveis, seja mais capacidade de processamento ou mais memória instalada. Para garantir esta característica ao código, foram utilizados recursos para recuperar os dados da GPU instalada e, caso exista mais de uma, dividir as tarefas entre elas.

As informações levadas em consideração são principalmente a quantidade de GPUs presentes e a versão do CUDA que cada uma pode executar. De posse destas informações, é possível verificar se os dados podem ser alimentados em três dimensões. Caso não seja possível, uma conversão para duas dimensões será necessária.

Caso não seja necessário realizar a transformação das dimensões, há a economia de uma etapa de cálculo para distribuir os cubos entre os núcleos da GPU, já que o *software* será alimentado com matrizes de três dimensões. Caso contrário, cálculos adicionais devem ser realizados de forma a mapear os índices dos *voxels* da imagem em uma matriz com uma dimensão a menos.

Inicialmente, para o desenvolvimento desta função foi elaborado um protótipo em CUDA/C para ser executado no MATLAB utilizando a interface MEX, a fim de validar o processamento das imagens e facilitar a depuração inicial do código.

Após verificados os dados produzidos do processamento da biblioteca MATLAB, foram retirados os códigos específicos da plataforma, de forma a criar uma função do

sistema operacional Microsoft Windows com código padrão C, utilizando somente as bibliotecas definidas pelo ANSI C e pelo padrão CUDA.

O algoritmo distribui os cubos formados por oito *voxels* das imagens entre as funções CUDA e os classifica de acordo com as tabelas encontradas na bibliografia (SUH e KIM, 2014).

A biblioteca dinâmica em C possui três etapas distintas. A primeira é desenvolvida em C e realiza a transferência dos dados do *host* para o *device*. A segunda é a chamada da função CUDA e, finalmente, a transferência dos dados processados do *device* para o *host*, com a limpeza da memória e retorno para o *software* C#.

Para a correta invocação da função CUDA o *host* deve definir dois valores, um denominado tamanho da grade (*grid size*) e outro denominado tamanho do bloco (*block size*). Estes números determinam, respectivamente, em quantas parcelas os dados devem ser divididos e a quantidade de *threads* utilizadas para o processamento de cada parcela. Estes números são obtidos a partir das dimensões da matriz criada pelas imagens tomográficas nos eixos X, Y e Z.

Para a validação do código elaborado em CUDA/C, foram utilizados testes abertos – testes caixa branca – verificando o valor das variáveis em tempo de execução. Para esta atividade, foi utilizado o depurador específico para GPU. Por intermédio do qual foi possível verificar se os dados tanto de saída quanto de entrada eram compatíveis com o que se espera do processamento. Verificando os valores que o módulo em C# passava para a função executando na GPU com o depurador da ferramenta de desenvolvimento Microsoft Visual Studio e em uma outra execução do *software*, utilizou-se a depuração do NSIGHT para verificar a consistência dos dados.

Para a validação dos dados de saída foram realizados dois testes. O primeiro similar ao teste para verificar os dados de entrada. Foram utilizados os dois depuradores para verificar se os bytes dos dados mantinham a consistência entre os módulos C# e CUDA/C.

Em outra execução do código foram comparados os dados retornados pelo código CUDA/C com os retornados na execução da biblioteca do MATLAB. Por meio deste procedimento foi possível verificar que ambos eram semelhantes, portanto o processamento é correto.

5. DISCUSSÃO DOS RESULTADOS

Este capítulo apresenta uma revisão do problema e da justificativa para a elaboração do trabalho na forma em que foi elaborado. Também serão revisitadas as dificuldades encontradas no desenvolvimento do aplicativo que o sistema proposto resolve.

Em seguida, é revisitado o sistema e são comparados tempos de execução e consumo de memória, em todas as implementações do algoritmo *Marching Cubes*, comparando o desempenho, tanto em relação ao consumo de memória quanto ao tempo de processamento e as dificuldades encontradas para a elaboração do *software*.

Posteriormente é realizada a compilação das dificuldades encontradas para a elaboração do aplicativo e é justificado por que foi escolhido este método para a solução do problema apresentado.

É explicada a paralelização do algoritmo, e como se encaixa o modelo de processamento da GPU no desenvolvimento do *software* em questão, bem como o benefício que este componente traz à resolução do problema.

5.1. JUSTIFICATIVA DO TRABALHO

O propósito do trabalho desenvolvido foi utilizar imagens tomográficas de um corpo de prova de concreto para realizar a reconstrução de um modelo computacional tridimensional a fim de utilizá-lo em ferramentas de simulação mecânica para ensaios não destrutivos.

A elaboração de um aplicativo que realizasse esse procedimento justifica-se pela necessidade de tratamento de um volume de dados maior que outras soluções analisadas são capazes de processar.

Inicialmente foi utilizada a implementação do *Marching Cubes* do VTK, mas após 40 minutos de execução, o aplicativo abortava o processamento com um erro de falta de memória. Para evitar esse erro, a solução era a diminuição da quantidade de fatias

tomográficas, a conjunto inferior a 2% da quantidade total de fatias que deveriam ser processadas (10 das 585 fatias tomográficas).

Outra solução para a geração da malha era a redução do número de triângulos. Entretanto, para que esta metodologia surtisse resultado, a redução tinha que ser da ordem de 95%, o que causava uma perda de definição no sólido que poderia comprometer os resultados da simulação pelo software de ensaios não destrutivos.

Para atender às demandas citadas anteriormente, foi realizada a implementação de uma solução de *software* que implemente de forma massivamente paralela o algoritmo *Marching Cubes* a fim de diminuir o tempo de processamento. Em adição a essa vantagem, o código implementado ainda deveria realizar a administração de memória com o objetivo de evitar exceder o espaço disponível para dados.

A forma escolhida para implementar este algoritmo foi, seguindo a linha de trabalho anteriormente desenvolvido pela equipe do projeto em que este trabalho se insere, a elaboração de uma solução que utiliza a GPU para reduzir o tempo necessário para a reconstrução tridimensional do sólido virtual através do processamento paralelo.

Para o gerenciamento de memória, foi utilizada a CPU, que se encarrega de criar espaços tridimensionais com um número menor de fatias tomográficas que o total e posteriormente unir os resultados dos processamentos, criando a malha tridimensional virtual correspondente ao sólido real inteiro, permitindo então o ensaio não destrutivo do corpo de prova como um todo e com o máximo de definição possível.

5.2. RECONSTRUÇÃO DA ISOSSUPERFÍCIE

Para a reconstrução das isossuperfícies, foi utilizado um conjunto crescente de fatias tomográficas, inicialmente composto de 10 fatias, em seguida 20, 50, 100, 200, 300, 400, 500 fatias e, finalmente, do conjunto tudo. Este procedimento foi adotado de forma a avaliar o tempo de processamento e realizar a verificação visual das superfícies.

A Figura 18 ilustra a reconstrução realizada utilizando a quantidade arbitrária de 20 fatias e com o isovalor 20.000, típico para o agregado. Na figura verifica-se que a isossuperfície corresponde às pedras do agregado, com o interior oco.

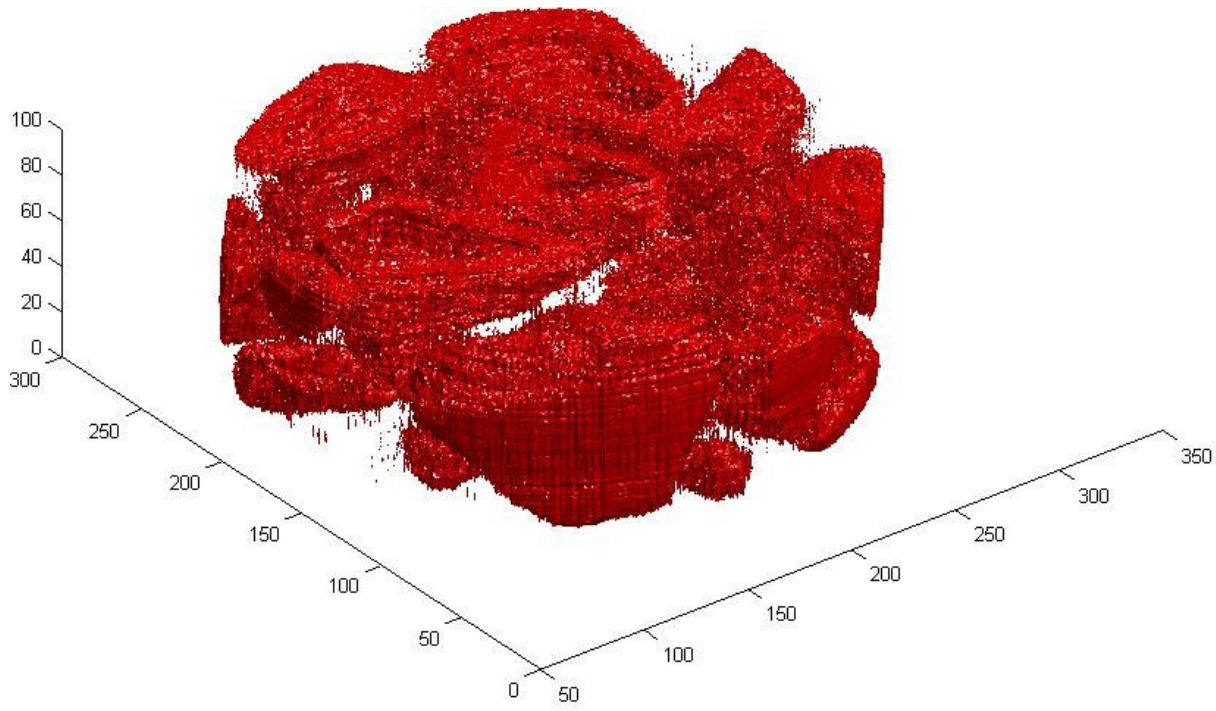


Figura 18: Reconstrução realizada utilizando vinte fatias e o isovalor típico para o agregado. Fonte: o autor.

A Figura 19 representa as mesmas fatias tomográficas, porém com o isovalor 50.000, específico do concreto. Na figura é possível visualizar o perímetro do testemunho de concreto e as bolhas de ar dentro da amostra.

Nota-se que, como esperado, somente a superfície do objeto foi reconstruída, desprezando o seu interior. Este comportamento já era esperado, uma vez que o algoritmo é para reconstrução das isossuperfícies ou seja, para representar a fronteira entre diferentes materiais que compõem a amostra.

As reconstruções das Figuras 18 e 19 foram feitas com uma quantidade pequena o suficiente de fatias tomográficas para ser possível ainda fazer a reconstrução unicamente na GPU, sem ser necessário a utilização do código de controle da quantidade de memória utilizada.

Para reconstruções utilizando mais fatias tomográficas, a quantidade de memória necessária é maior que o limite utilizável pelo *device*, portanto é necessária uma forma de melhor administrar os recursos de *hardware* disponíveis. Para este fim foi implementado um código que divide o domínio formado por todas as fatias tomográficas em subespaços que possam ser processados de forma paralela.

O módulo implementado retira de forma ordenada as fatias tomográficas do

espaço de memória do *host* a insere no *device* uma quantidade de fatias proporcional à quantidade de memória disponível, de forma a evitar erro de estrapolação de memória. Tão logo o processamento termine na GPU, o resultado é armazenado no espaço de memória da CPU, que se encarrega de enviar outras fatias para o processamento.

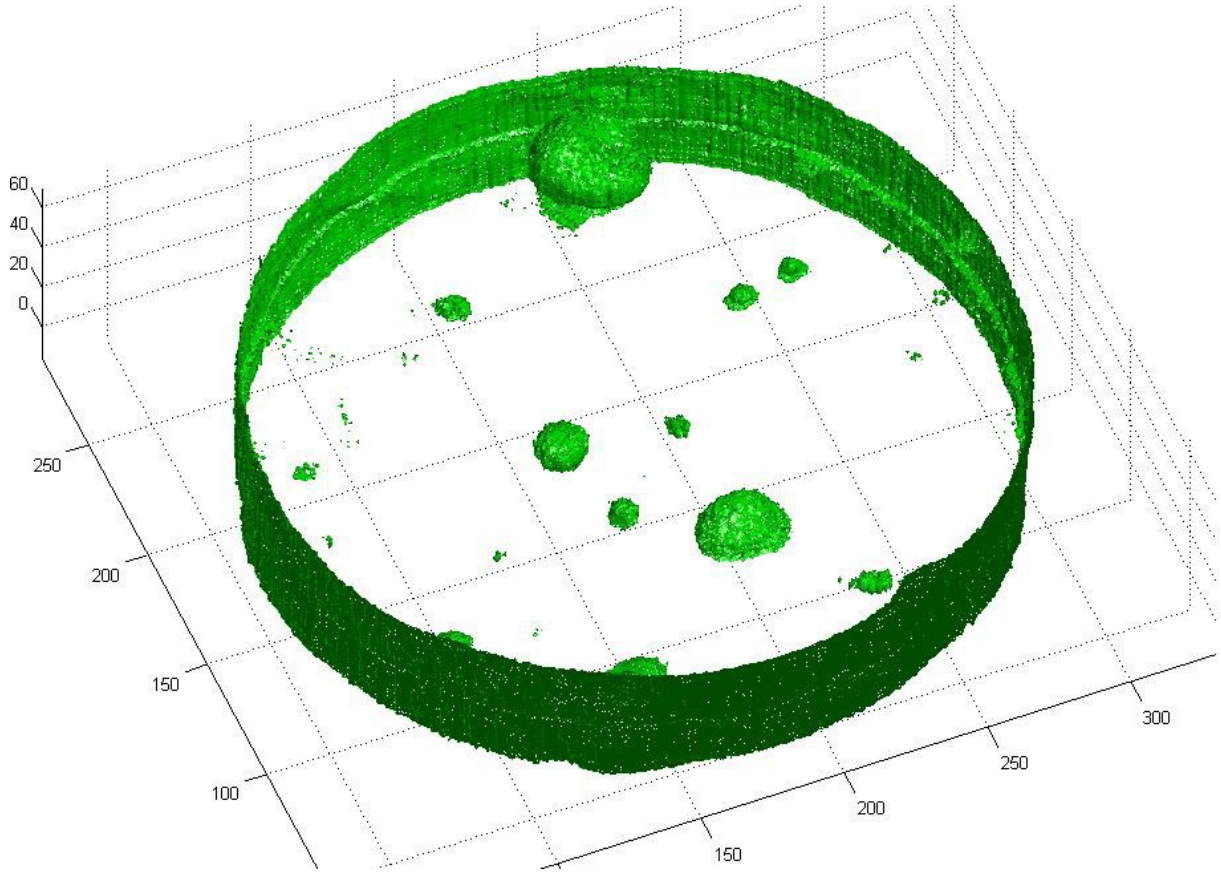


Figura 19: Reconstrução de vinte fatias tomográficas com o isovalor específico para o concreto.
Fonte: o autor.

Ao armazenar o retorno da nova etapa de processamento, o aplicativo faz a ligação entre os sólidos virtuais, de forma que as isossuperfícies reconstruídas formem um modelo contínuo, evitando que haja uma divisão no modelo que não corresponda à realidade. Este defeito poderia impactar de forma séria no procedimento de simulação, já que a ferramenta de *software* consideraria dois corpos distintos, em vez de apenas um.

Para validação, foi utilizado o conjunto inteiro de fatias tomográficas para a reconstrução da isossuperfície. Na Figura 20 mostra-se a reconstrução utilizando todas as fatias tomográficas com o isovalor típico para o agregado.

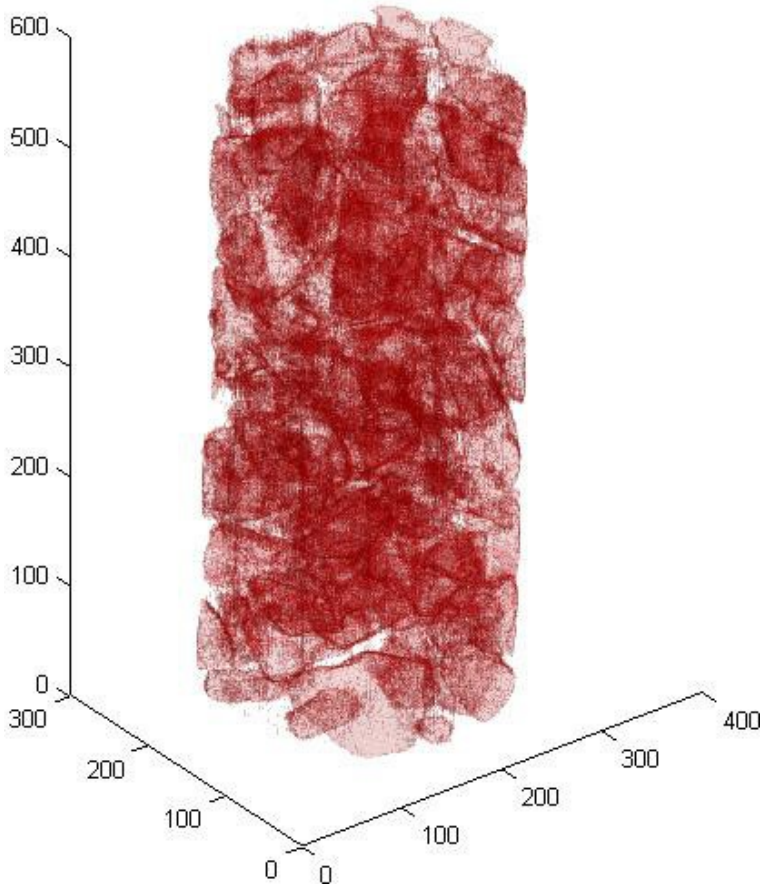


Figura 20: Reconstrução do conjunto completo de fatias tomográficas, com o valor típico para o agregado. Fonte: o autor.

É possível verificar na Figura 20 que não existe divisão no modelo, verificando que é realizada a fusão dos modelos após o processamento pelo *device*. Para melhorar a visualização das informações, foi aplicado sobre o modelo tridimensional uma transparência, de forma a permitir a visualização dos detalhes do agregado.

Finalmente, foi realizada a reconstrução do conjunto completo de tomografias utilizando o isovalor típico do concreto. O resultado é mostrado na Figura 21. Na Figura em questão é possível notar que o cilindro original foi reconstruído em sua totalidade.

As reconstruções das Figuras 20 e 21 confirmam que é possível utilizar o aplicativo desenvolvido para a extração de isossuperfícies a partir de imagens tomográficas por meio do processamento paralelo realizado em um contexto heterogêneo de maneira confiável e que é possível realizar a separação dos materiais com este método.

Para a simulação mecânica, portanto, é necessário somente realizar o

processamento com diferentes isovalores e utilizar os sólidos resultantes em conjunto com ferramentas apropriadas.

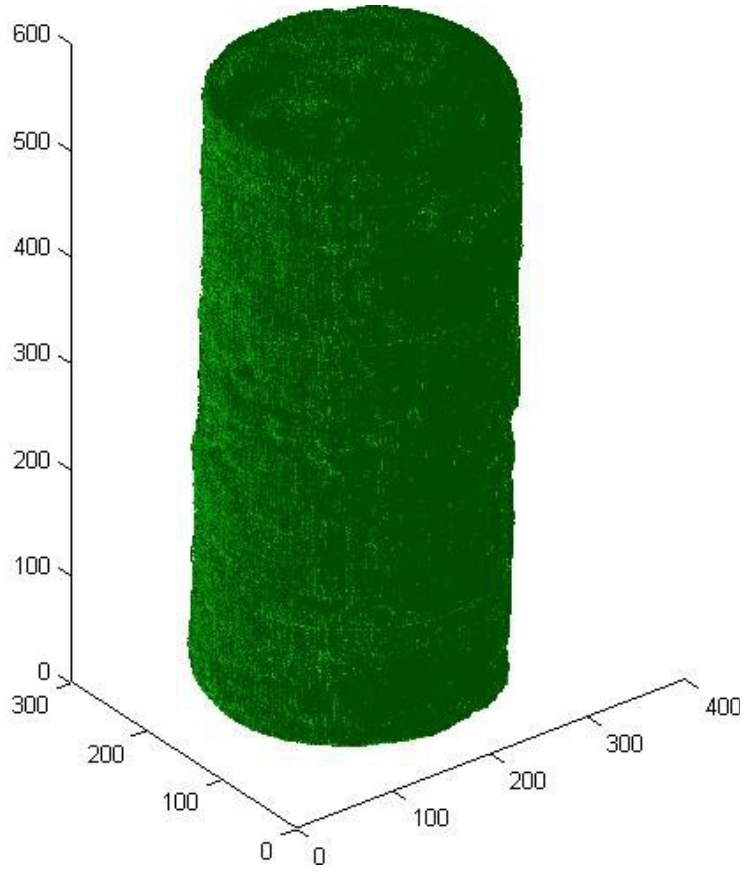


Figura 21: Reconstrução do conjunto completo de tomografias, utilizando o isovalor típico do concreto. Fonte: o autor.

A Figura Figura 22 mostra um detalhe da isossuperfície poligonal gerada pelo código implementado.

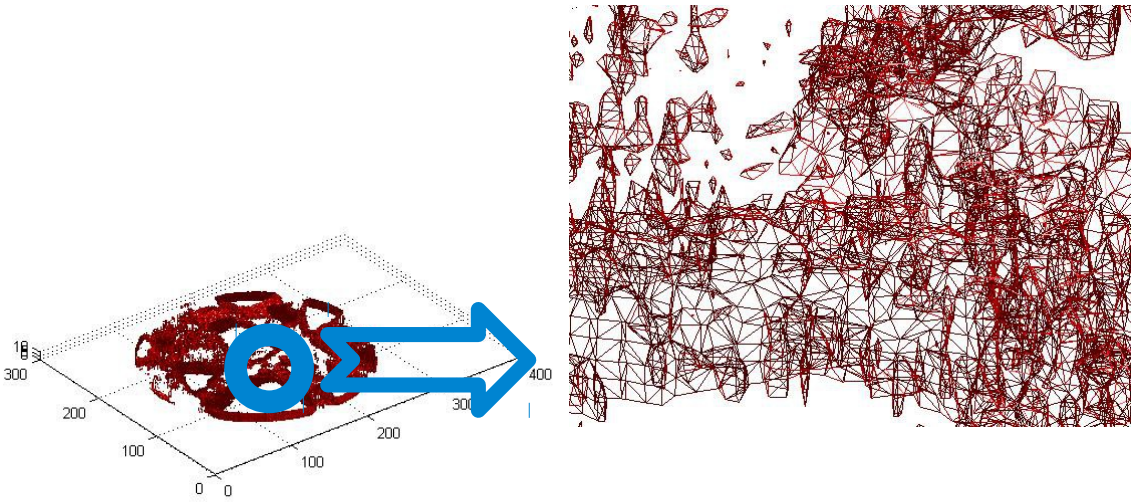


Figura 22: Detalhe dos triângulos gerados na isossuperfície poligonal. Fonte: o autor.

5.3. PROTOCOLO EXPERIMENTAL

A metodologia utilizada para a elaboração do algoritmo paralelo foi uma implementação em estágios, com um número de versões do código, cada uma utilizando uma técnica diferente de desenvolvimento de *software* e verificando a redução do tempo de processamento do código paralelo em relação ao código convencional, executado sequencialmente e, portanto, sem fazer uso de paralelismo.

O processo de validação do resultado do processamento – a isossuperfície poligonal – e a verificação da fidelidade em relação ao sólido real foi incorporado ao processo de desenvolvimento.

A primeira implementação do algoritmo foi realizada de forma fiel à bibliografia (SUH e KIM, 2014), salvo pela entrada de dados, que foi adaptada para a leitura das imagens tomográficas. Essa implementação foi utilizada como referência para as futuras implementações, uma vez que já havia sido validada na bibliografia.

Foi realizada a reconstrução com 3 conjuntos aleatórios de 50 fatias extraídos do conjunto total de 585 fatias. Os resultados das reconstruções fornecidas pelo algoritmo serial MATLAB obtidos nessa etapa foram salvos em matrizes da própria ferramenta para a comparação com as outras versões.

Em seguida foram executadas reconstruções com os mesmos conjuntos

utilizando as outras implementções – primeiro a vetorizada MATLAB e depois a massivamente paralela na GPU implementada em CUDA.

Para validar as versões do código, foram subtraídas as matrizes a serem comparadas da matriz de referência, quanto mais próximo da matriz nula o resultado dessa subtração, mais próxima a isossuperfície está da referência.

O resultado da subtração entre o código desenvolvido em script do MATLAB e o código MATLAB vetorizado foi, de fato, a matriz nula, mostrando que a consistência do processamento se manteve.

Para a implementação em CUDA, a verificação das matrizes de topologia resultou na matriz nula, enquanto a da matriz de vértices resultou em um erro na ordem de 10^{-6}, ou seja, um erro na ordem de 10^{-6} milímetros. Esta grandeza é aceitável, comparando com o tamanho da amostra, de 40 centímetros.

5.4. TEMPO DE EXECUÇÃO DO PROTÓTIPO MATLAB

Para a verificação da evolução do tempo de execução entre o código serial desenvolvido em script MATLAB, código serial utilizando o recurso de vetorização do MATLAB e o código paralelo implementado em CUDA/C foi utilizado o profiler – um aplicativo especializado em medir tempo de processamento – presente no MATLAB.

Inicialmente foi utilizado o conjunto total de tomografias e, deste universo de 585 imagens, foram escolhidos arbitrariamente subconjuntos para alimentar o código e verificar o tempo de processamento. Esta metodologia foi utilizada para verificar a progressão do tempo de processamento em relação à quantidade de dados fornecidos para processamento, uma vez que esta correspondência não é necessariamente linear (SEDGEWICK e FLAJORLET, 2013).

Os tempos de processamento medidos com o auxílio do profiler são mostrados na tabela 3. É importante notar que para a implementação serial do algoritmo em MATLAB foi realizada a medição com os conjuntos de 10, 20, 50 e 100 fatias, etapa na qual o objetivo das medições foi alcançado – demonstrar a ordem de grandeza do aumento do tempo de processamento do algoritmo serial não otimizado.

A tabela 3 permite verificar que o código MATLAB serial não só é o que possui o

maior tempo de processamento comparado com os outros algoritmos implementados — como esperado — como também tem a evolução mais íngreme em comparação com as outras medições realizadas, iniciando em aproximadamente de dez minutos para 10 fatias e indo para aproximadamente doze horas com um conjunto dez vezes maior, mas que ainda é quase um sexto do conjunto total de imagens.

Tabela 3: Tempos de processamento em segundos para as implementações do *Marching Cubes.*

Quantidade de fatias tomográficas	Código MATLAB paralelo	Código MATLAB vetorizado	Código GPU
10	644,27	0,89	0,83
20	2127,22	1,87	1,57
50	11414,31	4,22	4,79
100	42523,11	8,22	7,37
300	-	27,13	22,82
400	-	40,48	30,54
500	-	208,57	38,49
585 (Todas as imagens)	-	485,27	40,48

Esta característica da implementação serial é mais facilmente visualizada na Figura 23, em que se condensam os dados da Tabela 3.

Nota-se que o tempo de processamento serial é tão grande, que para visualizá-la é necessário usar uma escala que compromete a comparação entre os tempos das demais implementações.

Os tempos de execução das outras implementações iniciam-se muito similares, mas de acordo com o crescimento do conjunto de dados, o código vetorial se distancia do código paralelo implementado em GPU, conforme ilustra a Figura 24.

O código paralelo para GPU, nesta implementação precisou de um código protótipo para efetuar o controle de memória, uma vez que após 40 fatias tomográficas o aplicativo forçava o término da execução devido à falta de memória no *device*.

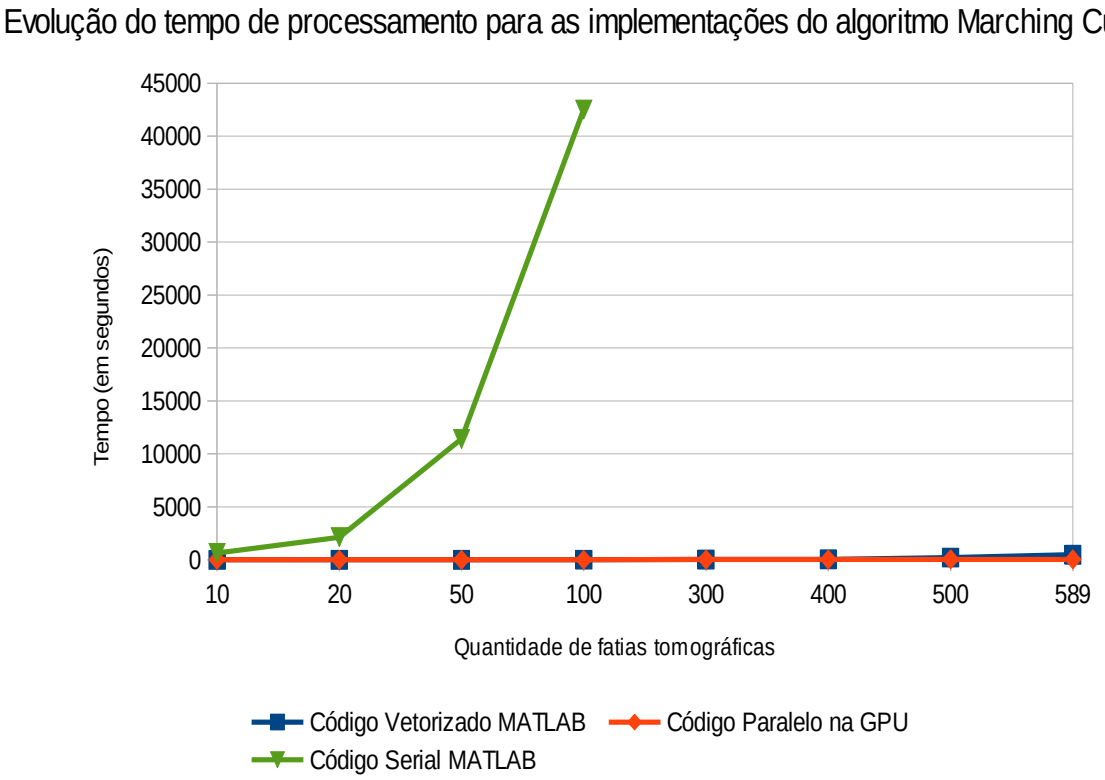


Figura 23: Evolução do tempo de processamento em relação ao número de imagens processadas.

Para fins de completude, a Figura 25 mostra a curva de evolução de tempo de processamento da implementação do algoritmo *Marching Cubes* em CUDA/C paralelizada na GPU.

Apesar do tempo de processamento dos algoritmos seriais obedecerem uma curva polinomial de terceira ordem, conforme esperado (SEDGEWICK e FLAJORLET, 2013), a evolução do algoritmo paralelo não corresponde à função linear, como era previsto.

Esta discrepância se dá pelo fato de ser necessária a colocação de código auxiliar para tratar o excesso de memória utilizado e é agravada pelo fato do código ter sido implementado em linguagem de script do MATLAB, o que faz com que o tempo adicional de interpretação da linguagem (*overhead* de processamento) seja em torno de 20% do tempo total de processamento.

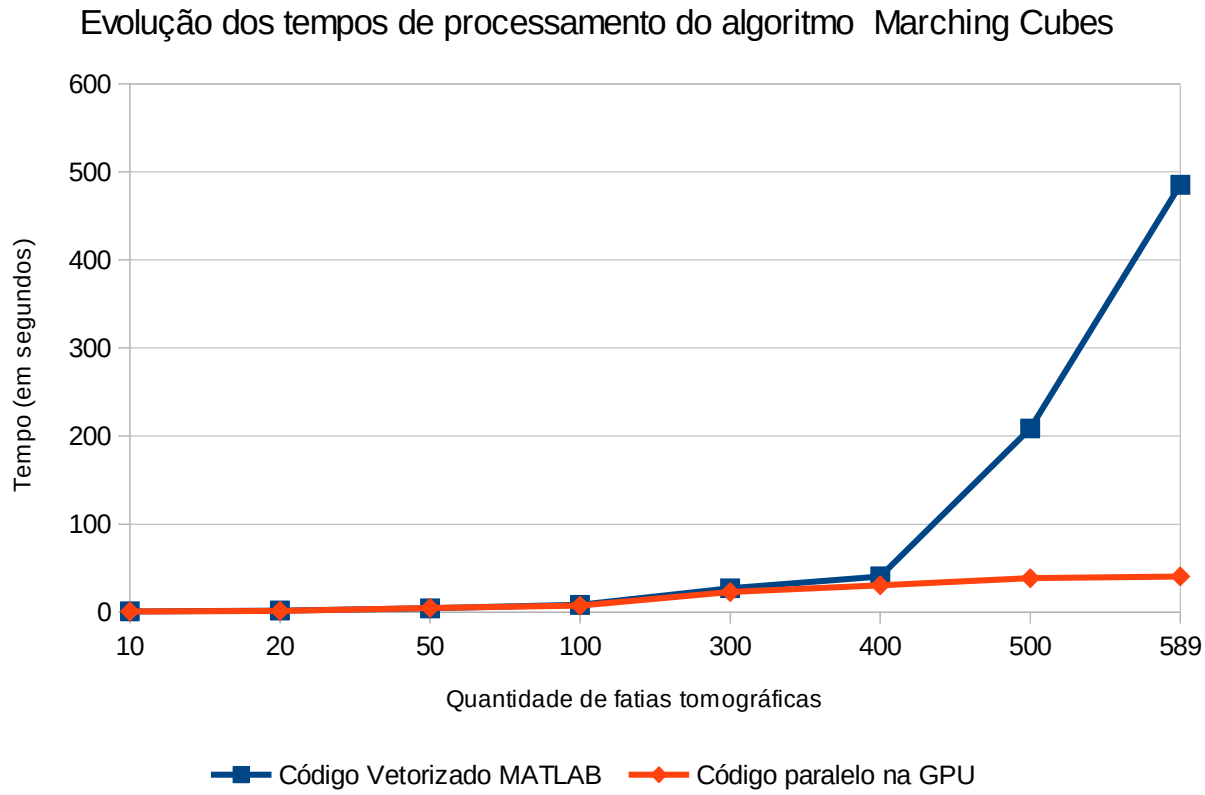
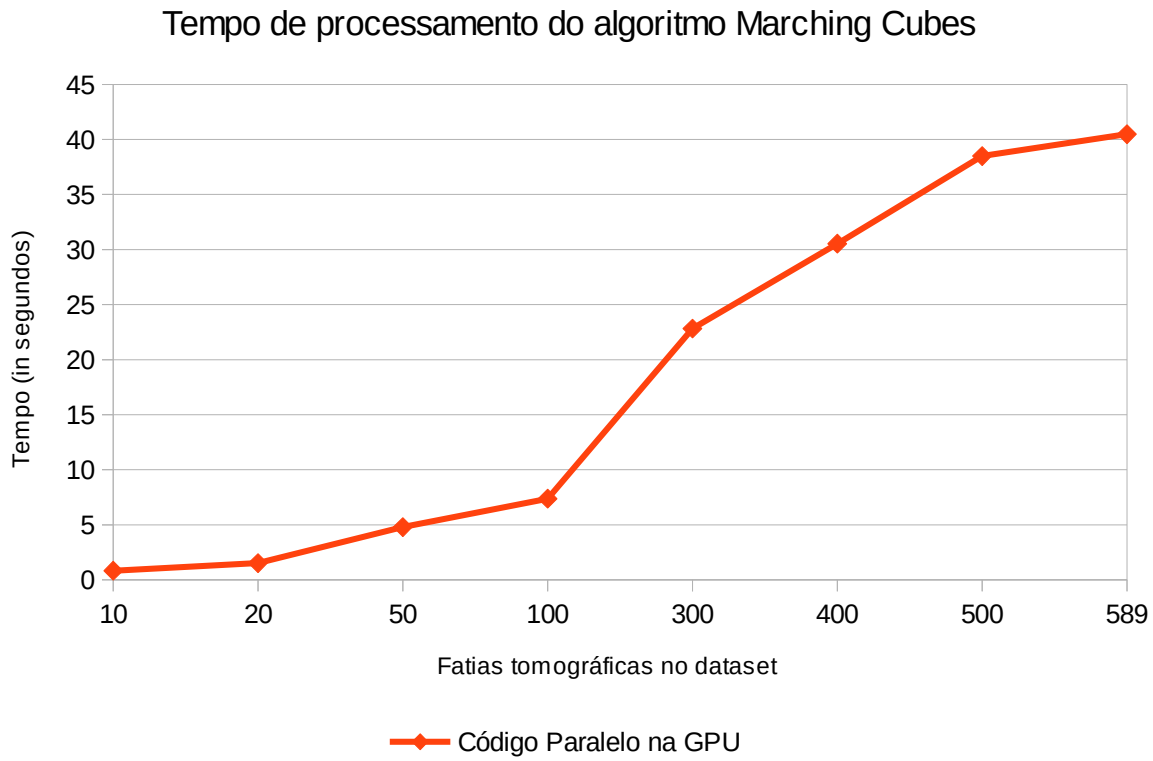


Figura 24: Tempos de processamento excluindo o código MATLAB serial.

Na Figura Figura 25, é possível notar a influência do código de gerenciamento de memória no tempo de processamento. Durante a medição, o *device* tinha como limite de memória a quantidade de 40 fatias tomográficas. Pode-se verificar que há a mudança de inclinação nas retas que formam o gráfico exatamente onde este limite se encontra – nos intervalos que contém valores múltiplos de 40.



FFigura 25: Tempo de processamento da implementação paralela em GPU do Marching Cubes.

5.5. CONSUMO DE MEMÓRIA

Apesar de não ser o foco principal do trabalho, foram analisados os dados sobre memória fornecidos pelo *profiler* do MATLAB a fim de avaliar o desempenho da implementação desenvolvida. Estes dados são importantes no sentido de realizar uma análise de desempenho tanto quanto à capacidade de processamento quanto ao consumo de memória.

Para realizar este comparativo, foi medido o consumo de memória entre o algoritmo vetorizado do MATLAB e a implementação paralela em CUDA com código auxiliar em MATLAB. Estes resultados podem ser melhorados com a transcrição do código de gerenciamento de memória do *device* pelo *host* utilizando a linguagem C.

Cabe aqui ressaltar que na implementação responsável por administrar a memória na implementação em CUDA há código protótipo, conforme já descrito anteriormente.

Inicialmente, foi verificada a quantidade de memória alocada pelo código em

execução. Esta memória é utilizada para o armazenamento de informações durante o processamento do aplicativo, contabilizando-se dados temporários, que serão descartados ao final do processamento, e os dados de retorno, que serão preservados para futura aplicação. A comparação de consumo de memória entre as implementações é ilustrada na Figura 26.

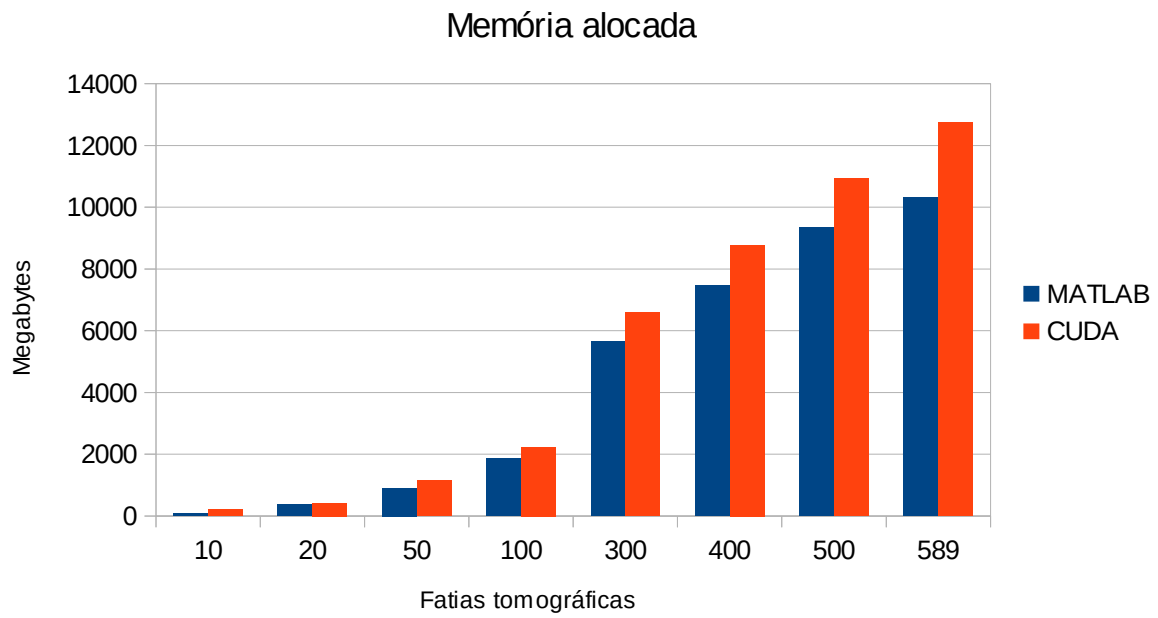


Figura 26: Memória alocada durante a execução das versões do algoritmo *Marching Cubes*.

Nota-se que a memória alocada pelo código paralelo é maior que a do código vetorizado MATLAB correspondende. Este fato se deve à implementação do código protótipo para o tratamento da memória ser menos eficiente que o introduzido no MATLAB pelo processo automatizado de vetorização. A Figura 27 ilustra outra característica do consumo de memória.

Nota-se que a evolução do pico de memória do código vetorizado é mais lenta que a do código paralelo, o que demonstra que o código rodando na GPU tem um consumo de memória maior que o vetorizado.

Estes resultados demonstram que, apesar da aceleração obtida no tempo de execução do aplicativo, o gerenciamento de memória ainda necessita de mais

desenvolvimento, mas que pode ser melhorado em versões futuras do aplicativo, com a melhor utilização da memória do *device* em detrimento à do *host*.

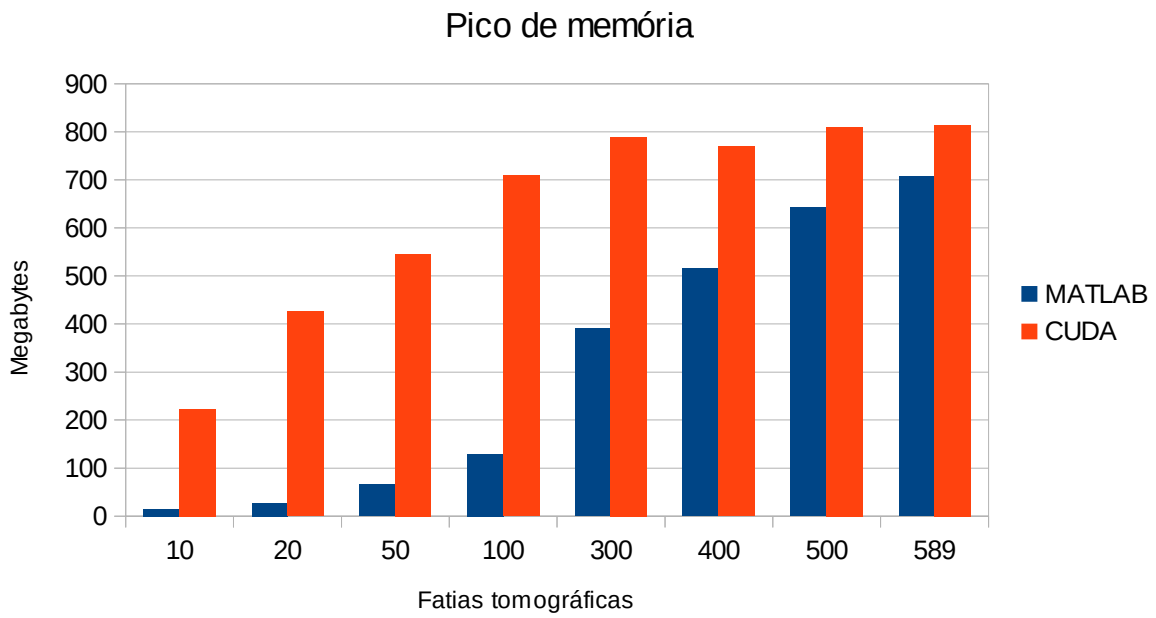


Figura 27: Pico de memória durante a execução.

As métricas utilizadas para verificar o desempenho do algoritmo massivamente paralelo demonstram que, apesar de o *device* ser o encarregado da maior parte das tarefas, o *host* ainda é necessário durante a execução da aplicação desenvolvida.

Esta característica mostra a necessidade de gerenciamento de memória, através de código auxiliar que executa de maneira serial na CPU, garantindo que o *device* não incorra no erro de falta de dados para processamento ou em extrapolação de memória.

6. CONCLUSÃO

Por este trabalho pôde-se constatar que não é trivial a implementação de algoritmos massivamente paralelos utilizando a GPU, apesar do ganho em tempo ser bastante significativo.

Este fato é decorrente da complexidade decorrente de necessidade de ferramentas de *software* específicas para o desenvolvimento e compilação de código. Para a depuração do código compilado, deve ser utilizado outro conjunto de ferramentas, que em alguns casos não fornecem informações precisas sobre erros de execução, levando a conclusões errôneas quanto à causa de problemas de solução trivial.

Outro ponto que agrega complexidade ao desenvolvimento para a GPU é a necessidade de monitoramento constante do desempenho por meio da utilização de um *profiler* – uma ferramenta que provê informações sobre a utilização do hardware e possíveis entraves à solução do problema.

A GPU é um componente que pode tratar um grande volume de dados muito rapidamente e esta característica é notada pela evolução do tempo de processamento, que sofreu um decréscimo muito grande para grandes volumes de dados.

Esta característica se baseia na grande quantidade de núcleos de processamento que compõem o microprocessador. Entretanto, deve-se ter em mente que a escolha errada do problema a ser processado ou a escolha de um algoritmo ineficiente pode resultar em um código paralelo com tempo de execução maior que o do código serial.

Este problema pode originar do fato que o *device* é um componente do *host*, sendo necessário tempo para a transmissão de dados e, caso o problema não seja paralelizável, a execução em uma parcela pequena da GPU pode demorar mais tempo que realizar a execução na CPU.

Além do mais, o desenvolvimento de código para a GPU é um procedimento difícil, com necessidade tanto de Hardware quanto de *Software* específicos. E a catalogação e agrupamento do ferramental para desenvolvimento de algoritmos massivamente paralelos pode ser considerado um dos maiores frutos do projeto desenvolvido.

A utilização da GPU (*Graphics Processing Unit* - Unidade de Processamento

Gráfico) permitiu também o ganho de velocidade no processamento. Inicialmente para o processamento das imagens eram necessárias mais de doze horas. Com a aplicação da técnica de paralelização do código o tempo caiu para pouco mais que 40 segundos.

6.1. TRABALHOS FUTUROS

O trabalho apresentado não só valida o conceito que o processamento paralelo com o auxílio da GPU é uma ferramenta importante para a elaboração de aplicativos mais eficientes, reduzindo o tempo de processamento total para grandes quantidades de dados sem perder precisão na resposta, como também cria uma ferramenta funcional para a geração de modelos tridimensionais confiáveis para simulações mecânicas não destrutivas.

As seguintes oportunidades de melhora do código desenvolvido podem ser exploradas por trabalhos futuros, utilizando a base sólida apresentada para implementar funcionalidade que tanto não estavam no escopo do trabalho desenvolvido quanto apareceram posteriormente, como subprodutos da implementação efetuada.

O método para gerenciamento de memória utilizado pode ser reimplementado utilizando a linguagem C, na porção executada no *host* da função CUDA. Esta modificação visa a redução ainda maior do tempo de processamento, substituindo-se o código protótipo MATLAB por uma linguagem de melhor desempenho.

Uma etapa de pré-processamento para a determinação de quais porções do espaço de fatias tomográficas devem ser tratadas também é necessária. Este tratamento pode ser feito por meio da determinação do volume em que existem dados válidos para o tratamento, – um *Bounding Box* – evitando o desperdício de processamento com dados irrelevantes.

Outra etapa que pode ser implementada é a contagem de *voxels* cujo valor está acima do definido. Esta quantidade deve ser utilizada posteriormente para alocar somente a memória necessária para a execução de determinado volume.

A implementação de um sistema de gerenciamento de memória mais sofisticado é necessário também. O *Device* deve verificar se no *Host* há memória o suficiente para a

transferência dos dados e, caso negativo, realizar a divisão do espaço em subespaços menores, que possam ser manejados. Esta abordagem necessita também de um método pelo qual os subespaços serão alimentados de forma iterativa pelo aplicativo no código paralelo.

Durante o desenvolvimento deste trabalho, bibliotecas desenvolvidas por terceiros foram aperfeiçoadas e ganharam suporte inclusive do fabricante. Algumas destas bibliotecas lidam com a complexidade de interface entre o CUDA/C e a plataforma .NET, utilizada pelo C#. É possível que os referidos projetos permitam uma integração mais estreita entre a utilização dos recursos da GPU com o aplicativo desenvolvido e sua adoção pode suprimir muito do código elaborado para a comunicação entre os ambientes gerenciado e não-gerenciado, fator que justifica a adoção de tais facilidades em versões futuras do aplicativo.

REFERÊNCIAS BIBLIOGRÁFICAS

ARCHIRAPATKAVE, V.; SUMILO, H.; SEE, S. C. W.; ACHALAKUL, T. GPGPU Acceleretion Algorithm for Medical Image Reconstruction. Parallel and Distributed Processing with Applications, **Proceedings**..., Busan: Institute of Electrical and Electronic Engineering, p. 41-46, mai. 2011.

BEHNIA, A.; CHAI, H. K.; YORIKAWA, M.; MOMOKI, S.; TERAZAWA, M.; SHIOTANI, T. Integrated non-destructive Assessment of Concrete Structures Under Flexture by Acoustic Emission and Travel Time Tomography. **Construction and Building Materials**, v. 67, p. 202-215, 2014.

CIRNE, M., V., M.; PEDRINI, H. Marching Cubes Technique for Volumetric Visualization Accelerated with Graphics Processing Units. **Journal of the Brazilian Computer Society**, v. 19, n. 3, p. 223-233, 2013.

COOK, S. **CUDA Programming A Developer's Guide to Parallel Computing with GPUs**. Waltham: Elsevier, 2013.

DYKEN, C.; ZIEGLER, G.; THEOBALT, C.; SEIDEL, H. High-speed Marching Cubes using HistoPyramids. **Computer Graphics Forum**, Malden, v. 27, n. 8, p. 2028-2039, 2008.

FARBER, R. **CUDA Application Design and Development**. Waltham: Elsevier, 2011.

FARBER, R. **CUDA Application Design and Development**. Elsevier: Waltham, 2013.

FOLEY, J. D.; DAM, A. van; FEINER, S. K.; HUGHES, J. F. **Computer Graphics: Principles and Practice in C**. 2. ed. New York: Addison-Wesley, set. 1997.

FREEMAN, A. **Pro .NET 4 Parallel Programming in C#**. Apress: New York, 2010.

GARBOCZI, E. J. Three-dimensional Mathematical Analysis of Particle Shape Using X-ray Tomography and Spherical Harmonics: Application to Aggregates Used in Concrete. **Cement and Concrete Research**, v. 32, n. 10, p. 1621-1638, oct. 2002.

GEBALI, F. **Algorithms and parallel computing**. New Jersey: Wiley & Sons, 2011.

HANSEN, C. D.; HINKER, P. Massive Parallel Isosurface Extraction. In: IEEE Conference in Visualization. **Proceedings**... , Boston: Institute of Electrical and Electronic Engineering, 1992, p. 77-83.

HENESSY, J. L.; PATTERSON, D. A. **Computer Architecture: a quantitative approach.** 4. ed. São Francisco: Elsevier, 2007.

HOGENSON, G. **C++/CLI The Visual C++ Language for .NET**. Apress: New York, 2006.

HONG, Q.; WANG, B.; LI, Q.; LI, Y.; WU, Q. GPU Accelerating Technique for Rendering Implicitly Represented Vasculatures. **Bio-Medical Materials and Engineering**, v. 24, p. 1351-1357, 2014.

INTEL. **4th Generation Intel Core i3 Processors**. Disponível em: <http://ark.intel.com/products/family/75025>. Acesso em: 06/08/2014.

INTEL. **4th Generation Intel Core i5 Processors**. Disponível em: <http://ark.intel.com/products/family/75024>. Acesso em: 06/08/2014.

INTEL. **4th Generation Intel Core i7 Processors**. Disponível em:

<http://ark.intel.com/products/family/75023>. Acesso em: 06/08/2014.

INTEL. **Intel 64 and IA-32 Architectures Software Developer's Manual**. INTEL, 2014. Disponível em: <www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf>. Acessado em 07/08/2014.

KIRK, D. B.; HWU, W. W. **Programando para processadores paralelos: uma abordagem prática à programação de GPU.** Rio de Janeiro: Elsevier, 2011.

KITWARE. **Visualization Toolkit**. Disponível em: <www.vtk.org>. Acessado em 12/08/2014.

LORENSEN, W. E.; CLINE, H. E. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. **Computer Graphics**, New York, v. 21, n. 4, p. 163-169, jul. 1987.

MILI, M.; MAHMOUD, B.; AKIL, M.; BEDOUI, M. H. Hardware Parallel Architecture of a 3D Surface Reconstruction: Marching Cubes Algorithm. **International Journal of Circuits, Systems and Signal Processing**, v. 6, n 2, p. 143-150, 2012.

MALIK, M.; LI, T;. SHARIF, U.; SHAHID, R.; EL-GHAZAWI, T.; NEWBY, G. Productivity of GPUs under different programming paradigms. **Concurrency and Computation: Practice and Experience**, v. 24, p. 179-191, 2012.

MONTANI, C.; SCATENI, R.; SCOPIGNO, R. Discretized Marching Cubes. In: IEEE Conference in Visualization. **Proceedings**... , Washington: Institute of Electrical and Electronic Engineering, 1994, p. 281-287.

NEWMAN, T. S.; Yi, H. A Survey of the Marching Cubes Algorithm. **Computers & Graphics**, v. 30, n. 5, p. 854-879, out. 2006.

NVIDIA. **GeForce256**. Disponível em: <http://www.nvidia.com/page/geforce256.html>. Acessado em 07/08/2014.

NVIDIA. **NVIDIA GPU Programming Guide**. Disponível em: <http://developer.download.nvidia.com/GPU_Programming_Guide/GPU_Programming_Guide.pdf>. Acesso em: 06/08/2014.

NVIDIA. **NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110**. Whitepaper. Disponível em: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>. Acesso em: 06/08/2014.

NVIDIA. **Tesla GPU Accelerators for Servers**. Disponível em: <http://www.nvidia.com/object/tesla-servers.html>. Acesso em: 06/08/2014.

NVIDIA. **What is CUDA**. Disponível em: <http://developer.nvidia.com/what-cuda>. Acesso em: 06/08/2014.

NIELSEN, G. M. Dual Marching Cubes. In: IEEE Visualization, 2004, Austin. **Proceedings**... Picataway: Institute of Electrical and Electronic Engineering, 2004. p. 489-496.

PAN, T.; LLOYD, S. F. X-ray Tomography Based Microstrutural Model for Multi-Physical Analyses of Concrete. **Pavement, Structures, and Performance**, Shanghai: American Society of Civil Engineers, p. 100-108, mai. 2014.

PARHAMI, B. **Introduction to Parallel Processing Algorithms and Architectures**. New York: Kluwer Academic Publishers, 2002.

REED, D. **A Balanced Introduction to Computer Science**. Prentice Hall, 2008.

SANTOS, R. C. R.; GEUS, K. de; GODOI, W. C.; SCHEER, S. **Geometry extraction of high resolution CT studies from concrete structural samples: a massive parallel approach using a GPU architecture**, Fortaleza, 2014.

SEDGEWICK, R.; FLAJORLET, P. **An Introduction to the Analysis of Algorithms**. 2. ed. Addison-Wesley: Massachusetts, 2013.

SHREINER, D.; SELLERS, G.; KESSENICH, J.; LICEA-KANE, B. **OpenGL Programming Guide**. 8. ed. Addison-Wesley: Michigan, 2013.

STEIN, R. C.; PETKOVSKI, M.; ENGELBERG, D. L.; LEONARD, F.; WITHERS, P. J. Characterizing the Effects of Elevated Temperature on the Air Void Pore Structure of Advanced Gas-cooled Reactor Pressure Vessel Concrete Using X-ray Computed Tomography. **EPJ Web of Conferences**, v. 56, p. 04003-p.1 – 04003-p.7, 2013.

SUH, J. W.; KIM, Y. **Accelerating MATLAB with GPU computing**. Waltham: Elsevier, 2014.

SUZUKI, T.; MORII, T.; KAWAI, T. Damage Evaluation of Concrete Structure in Disaster Areas due to the Great East Japan Earthquake. **Niigata University Faculty of Agriculture Research Report**. v. 65, n. 2, p. 157-163, mar. 2013.

WRIGHT, Jr., R. S.; HAEMEL, N.; SELLERS, G.; LIPCHAK, B.; **OpenGL Superbible**. Michigan: Addison-Wesley, 2011.

ZHANG, H.; NEWMAN, T. Efficient Parallel Out-of-core Isosurface Extraction. IEEE Symposium on Parallel and Large-Data Visualization and Graphics, 2003, Seattle.

Proceedings... Picataway: Institute of Electrical and Electronic Engineering, 2003. p. 9-16.

ZIEGLER, G.; DYKEN, C. GPU-Accelerated Data Expansion for the Marching Cubes Algorithm. GPU Technology Conference. **GTC On-Demand**... San Jose, 2010. Disponível em: <on-demand.gputechconf.com/gtc/2010/presentations/S12020-GPU-Accelerated-Data-Expansion-Marching-Cubes-Algorithm.pdf>. Acessado em 07/08/2014.