UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS

Leonardo Maronezi Pizani

Estudo sobre métodos para aceleração de um algoritmo de reconstrução de imagens de tomossíntese digital mamária baseado em hardware GPU e programação CUDA

São Carlos

2020

Leonardo Maronezi Pizani

Estudo sobre métodos para aceleração de um algoritmo de reconstrução de imagens de tomossíntese digital mamária baseado em hardware GPU e programação CUDA

Monografia apresentada ao Curso de Engenharia Elétrica com Ênfase em Eletrônica, da Escola de Engenharia de São Carlos da Universidade de São Paulo, como parte dos requisitos para obtenção do título de Engenheiro Eletricista.

Orientador: Prof. Dr. Marcelo A. C. Vieira

São Carlos

2020

Ficha catalográfica elaborada pela Biblioteca Prof. Dr. Sérgio Rodrigues Fontes da EESC/USP com os dados inseridos pelo(a) autor(a).

Pizani, Leonardo Maronezi
P695e Estudo sobre métodos para aceleração de um algoritmo de reconstrução de imagens de tomossíntese digital mamária baseado em hardware GPU e programação CUDA / Leonardo Maronezi Pizani; orientador Marcelo Andrade da Costa Vieira. São Carlos, 2020.

Monografia (Graduação em Engenharia Elétrica com ênfase em Eletrônica) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2020.

1. GPU. 2. Programação paralela. 3. CUDA. 4. Tomossíntese digital mamária. I. Título.

Eduardo Graziosi Silva - CRB - 8/8907

FOLHA DE APROVAÇÃO

Nome: Leonardo Maronezi Pizani

Título: "Aceleração e Optimização de Código via Programação em GPU com CUDA"

Trabalho de Conclusão de Curso defendido e aprovado em 25/06/2020,

com NOTA 9,0 (NOVE, ZERO), pela Comissão Julgadora:

Prof. Associado Marcelo Andrade da Costa Vieira - Orientador - SEL/EESC/USP

Prof. Associado Evandro Luis Linhari Rodrigues - SEL/EESC/USP - Aposentado

Mestre Rodrigo de Barros Vimieiro - Doutorando - SEL/EESC/USP

Coordenador da CoC-Engenharia Elétrica - EESC/USP:
Prof. Associado Rogério Andrade Flauzino

AGRADECIMENTOS

Agradeço primeiramente a meus pais, por todo apoio e confiança demonstrados desde muito antes de eu ingressar na universidade.

Agradeço a meus amigos, por tornarem suportável mesmo as experiências mais estressantes. Agradeço especialmente ao Daniel e ao Rafael, com quem dividi minhas dúvidas e frustrações durante o desenvolvimento deste trabalho.

Agradeço ao Rodrigo, pelas dicas e pelos conhecimentos ofertados, sem os quais este trabalho não seria possível.

Agradeço por fim ao meu professor orientador, Marcelo, e a todos os outros professores e funcionários desta universidade, cujo trabalho contribui ativamente para a existência do ensino público superior de qualidade no país.

RESUMO

Pizani, L. M. **Estudo sobre métodos para aceleração de um algoritmo de reconstrução de imagens de tomossíntese digital mamária baseado em hardware GPU e programação CUDA** . 2020. 73p. Monografia (Trabalho de Conclusão de Curso) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2020.

Este trabalho tem como objetivo explorar algumas das funcionalidades do CUDA *toolkit*, utilizando uma *toolbox* de reconstrução de imagens de tomossíntese digital mamária desenvolvida para MATLAB. Foram testadas diferentes abordagens nesse processo, modificando o algoritmo empregado nos testes a fim de aumentar seu grau de paralelismo e permitir a aplicação de algumas das principais ferramentas disponibilizadas pelo CUDA *toolkit*, como *shared memory*, *texture objects* e *streams*. Como o algoritmo utilizados nos testes tem a função de gerar projeções 2D, foram aplicadas métricas de comparação de imagens em todos as projeções geradas utilizando as imagens do algoritmo original como controle, a fim de validar os resultados. Foram calculados os tempos médios de cada abordagem e os valores obtidos foram comparados para permitir uma análise de quais métodos apresentaram os melhores resultados e quais fatores mais influenciaram no tempo de execução. Com os testes foi possível concluir que, dependendo do algoritmo utilizado, a redução de tempo na execução do algoritmo pode chegar a mais de 180 vezes ao empregar CUDA. Os resultados mais significativos foram consequência das mudanças que aumentaram o grau de paralelismo do algoritmo, porém o uso de ferramentas mais específicas do CUDA *toolkit* também podem fazer uma diferença de até 1,8 vezes quando aplicadas nas condições certas.

Palavras-chave: GPU, programação paralela, CUDA, tomossíntese digital mamária.

ABSTRACT

Pizani, L. M. **Study on methods for accelerating an image reconstruction algorithm for digital breast tomosynthesis based on GPU hardware and CUDA programming**. 2020. 73p. Undergraduate Final Project, Sao Carlos School of Engineering, University of Sao Paulo, Sao Carlos, Brazil, 2020.

This work aims to explore some of the CUDA toolkit features, using the toolbox for reconstruction of digital breast tomosynthesis images developed in MATLAB. Different approaches were tested in this process, modifying the algorithm used in the tests to increase its degree of parallelism and allow the application of some of the main tools available by CUDA toolkit, such as shared memory, texture objects and streams. In order to validate this work, the accelerated projection images were compared against the results provided by the original toolbox. For each acceleration method, the average execution time was evaluated in order to choose the best algorithm, and also what are the most important factors that impact the acceleration procedure. Depending on the technique applied, the time reduction factor, in the algorithm execution, can reach more than 180 times, when using GPU with the CUDA toolkit. The higher speed improvements were achieved due to the higher parallelism applied in the algorithms. Moreover, specific CUDA techniques can also increase the speedup up to 1.8 times, when applied in the right manner.

Keywords: GPU, parallel programming, CUDA toolkit, digital breast tomosynthesis.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

1D	Uma Dimensão
2D	Duas Dimensões
3D	Três Dimensões
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DRAM	Dynamic Random Access Memory
FLOPS	FLoating-point Operations Per Second
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
NaN	Not a Number
PCIe	Peripheral Component Interconnect Express
PSNR	Peak Signal to Noise Ratio
RAM	Random Access Memory
SM	Streaming Multiprocessor
SSIM	Structural Similarity Index

SUMÁRIO

1 INTRODUÇÃO

1.1 Contextualização e Motivação

Em 1965 o cofundador da Intel, Gordon Earle Moore, montou um gráfico que descrevia a quantidade de transistores que era possível integrar em um único chip desde 1959, ano da invenção do circuito integrado, até então. Observando a tendência do gráfico Moore previu que a quantidade de transistores por chip dobraria a cada ano, previsão essa que veio a se tornar conhecida como Lei de Moore (Huang et al., 2010; Yao, 2004). Posteriormente essa previsão foi revisada e o período de tempo necessário para dobrar a quantidade de transistores em um chip foi alterada para dois anos (Denning and Lewis, 2017; McCool et al., 2012).

A Lei de Moore se mostrou uma estimativa razoável durante décadas, entretanto o aumento no número de componentes não foi o único responsável pelo crescimento exponencial na velocidade de processamento observada, sendo o aumento da frequência de chaveamento dos transistores, também chamada de frequência de clock, um fator igualmente relevante (Theis and Solomon, 2010; Ramanathan et al., 2015; Denning and Lewis, 2017). Durante muito tempo esses dois fatores foram suficientes para manter a elevada taxa de desenvolvimento tecnológico praticamente inalterada, porém ambos possuem limites físicos fundamentais.

Transistores menores necessitam de menos energia, mas isso requer que a tensão de operação diminua juntamente com o tamanho dos transistores, o que pode afetar sua operação, uma vez que é necessária uma tensão mínima para que o componente mude sua característica condutiva corretamente e represente o nível lógico que deveria (Theis and Solomon, 2010; Ramanathan et al., 2015). Além disso as altas velocidades de clock induzem grandes variações de corrente, o que implica em maiores correntes de fuga e em um consumo de energia que não diminui na mesma proporção que a dimensão dos transistores. Em outras palavras aumentar o número de componentes em um chip de área fixa, como um processador por exemplo, também aumenta seu consumo de energia e, como consequência, gera mais calor(Huang et al., 2010; Ramanathan et al., 2015; Yao, 2004). Esse aumento progressivo no consumo de potência e geração de calor dos processadores é chamado pela Intel de *power wall*.

Ao juntar as limitações da *power wall* com a latência de acesso a memória é possível entender porque a indústria passou a fixar a tensão de alimentação dos transistores e parou de simplesmente aumentar sua velocidade de chaveamento, mesmo que ainda continuem diminuindo seu tamanho (Theis and Solomon, 2010; Ramanathan et al., 2015). Sabendo da impossibilidade de manter o crescimento exponencial da velocidade de processamento

pelos meios supracitados foi necessário a adoção de um novo paradigma.

Uma opção muito estudada recentemente é a integração de chips tridimensionais, uma nova tecnologia que, em vez de integrar todos os transistores de um chip lado a lado em um mesmo plano, visa empilhar verticalmente vários desses planos. Essa tecnologia possibilita maior densidade de componentes por chip e novas organizações de circuitos, porém isso também resulta em um drástico aumento na dissipação de calor no componente, fator que já é preocupante mesmo nas tecnologias bidimensionais (Ahmed and Schuegraf, 2011; Loh et al., 2007; Huang et al., 2010).

Outra opção é diminuir os esforços em desenvolver máquinas com um único processador cada vez mais poderoso e concentra-los em desenvolver computadores com vários núcleos de processamento, e é exatamente isso que a indústria tem feito nos últimos anos (Adve et al., 2008). Praticamente todos os computadores modernos já se utilizam deste princípio, possuindo pelo menos um recurso paralelo em seu *hardware*, de fato chips *multicore* estão tão difundidos atualmente que até no simples ato de comprar um *smartphone* novo é comum se deparar com a escolha de processadores *dualcore*, *quadcore* e até *octacore* (McCool et al., 2012), embora nem todo mundo saiba o que essas terminologias significam. Os processadores *multicore*, como são chamados, podem não ser necessariamente mais rápidos que um modelo de um núcleo dos mais avançados, mas utilizando do princípio de programação paralela podem apresentar desempenho geral superior (Keckler et al., 2009; Geer, 2005).

A programação paralela consiste de dividir grandes tarefas em frações menores e mais simples, que são executadas de forma independente entre si, concomitantemente, e em diferentes núcleos de processamento. Desta forma diretivas de *loop*, por exemplo, dependendo de sua natureza, podem ter todos seus ciclos efetuados de uma única vez, poupando grande quantidade de tempo. Contudo poucas são as ferramentas que efetivamente foram desenvolvidas pensando em programação em paralelo (McCool et al., 2012).

As primeiras linguagens de máquina eram fundamentalmente seriais, ou seja, efetuavam tarefas consecutivas sequencialmente, até mesmo diretivas de *loop* eram apenas uma forma abreviada de executar tarefas sequenciais uma grande quantidade de vezes. Com o passar do tempo a serialização se arraigou no processo de programação de tal forma que a maioria dos recursos e dos conhecimentos sobre programação difundidos até hoje são essencialmente serializados, mesmo que o *hardware* utilizado possua alto grau de paralelismo (McCool et al., 2012).

Isso porém está mudando, recentemente o paralelismo vem sendo cada vez mais utilizado em vários domínios, incluindo computação gráfica, computação de alto desempenho e também em ambas combinadas (Adve et al., 2008). As chamadas GPUs (Graphics Processing Unit) são processadores altamente especializados em renderização de imagens, efetuando grandes quantidades de cálculos de ponto flutuante rapidamente, chegando até

a alcançar a casa dos GFLOPS (10^9 Floating-point Operations Per Second) (McClanahan, 2010; Shah and Yousaf, 2019). Uma vez que o cálculo da cor de um pixel na tela é independe da cor dos demais pixels, a paralelização das placas gráficas foi a escolha lógica, escolha essa que tem feito com que o desenvolvimento das GPUs acelerasse ainda mais que o previsto pela Lei de Moore (McClanahan, 2010; Brodtkorb et al., 2013).

De fato a tecnologia das placas gráficas cresceram tão rapidamente que tem também sido utilizada em computação de propósito geral juntamente com as CPUs (Central Processing Unit), recebendo assim o nome de GPGPU (General Purpose Graphics Processing Unit)(Brodtkorb et al., 2013). As GPGPUs não são a única alternativa para programar paralelamente de forma eficiente, mas tendem a ser as mais utilizadas, uma vez que placas gráficas já são facilmente encontradas em computadores atuais por padrão, não necessitando de um hardware completamente avesso ao cotidiano da maioria dos programadores, como é o caso das FPGAs (Field Programmable Gate Array), por exemplo (Brodtkorb et al., 2013).

Os maiores fabricantes de placas gráficas atualmente são dois, AMD e NVIDIA, porém a NVIDIA tem apresentado mais destaque no meio acadêmico devido a concepção de sua arquitetura de computação paralela própria, chamada CUDA (Compute Unified Device Architecture) (Brodtkorb et al., 2013).

A demanda por custo computacional vem crescendo rapidamente e é seguro dizer que vai continuar crescendo, principalmente no meio acadêmico, no qual novos algoritmos cada vez mais complexos são desenvolvidos a todo momento. Esses novos algoritmos necessitam ser cada vez mais velozes, uma vez que para serem refinados e validados são necessários inúmeros testes, do contrario todo o processo de desenvolvimento será lento e dispendioso.

Além disso, a simples aplicabilidade de um algorítimo pode depender de sua velocidade, como é o caso das aplicações médicas, já que muitas vezes os médicos e seus pacientes necessitam dos resultados de exames o mais rápido possível. O processo de reconstrução de tomossíntese digital mamária, por exemplo, composto por métodos de projeção, retroprojeção e técnicas de filtragem de ruído, já é conhecido há bastante tempo, mas devido ao seu alto custo computacional tardou a ser foco de pesquisas (Vimieiro, Rodrigo de Barros, 2019).

Os processos iterativos de reconstrução são lentos sequencialmente mesmo quando executados em CPUs modernas, mas acarretam em grandes benefícios no âmbito da saúde, uma vez que tem por objetivo gerar imagens para o rastreio do câncer de mama (Vimieiro, Rodrigo de Barros, 2019). Por esse motivo são ótimas opções para se acelerar usando programação paralela, e a tecnologia CUDA, além de permitir isso, proporciona uma experiência de programação familiar ao usuário, permitindo que ele programe em uma linguagem que já domine (Arora, 2012).

1.2 Objetivo

O presente trabalho visa investigar o funcionamento de algumas das diferentes ferramentas e métodos de optimização e aceleração de código que a NVIDIA disponibiliza através do CUDA *Toolkit*, utilizando como objeto de estudo uma *toolbox* de reconstrução de tomissíntese digital mamária que, em trabalhos anteriores, já foi testada em um modelo de programação serial. Contudo, como o objetivo deste trabalho é entender como empregar CUDA de forma efetiva, o único algoritmo utilizado nos testes foi o que gera as projeções, a fim de que, futuramente, os aprendizados deste trabalho possam ser aplicados para acelerar o processo de reconstrução por completo.

1.3 Organização da Monografia

Além deste capítulo o presente trabalho está dividido em mais 4 secções, que são respectivamente Teoria, Materiais e Métodos, Resultados e Conclusão.

O Capítulo 2, Teoria, apresenta de maneira detalhada os conceitos da programação paralela, bem como sua aplicação por meio das novas diretivas de C++ proporcionadas pela arquitetura CUDA. Nesse capítulo também são explicados os conceitos básicos a respeito de reconstrução de imagens de tomossíntese digital mamária, de forma breve e resumida, apenas a fim de orientar adequadamente o leitor e possibilitar melhor entendimento da secção de Resultados.

O Capítulo 3, Materiais e Métodos, descreve como foi a abordagem utilizada para efetuar a tradução dos códigos seriais previamente estudados em códigos paralelizados por meio de CUDA.

O Capítulo 4, Resultados, tem a finalidade de expor os resultados obtidos durante o decorrer do trabalho e discorrer a respeito deles.

Por fim no Capítulo 5, Conclusão, é apresentada uma visão geral de como se procedeu o trabalho e se ele alcançou as expectativas.

2 TEORIA

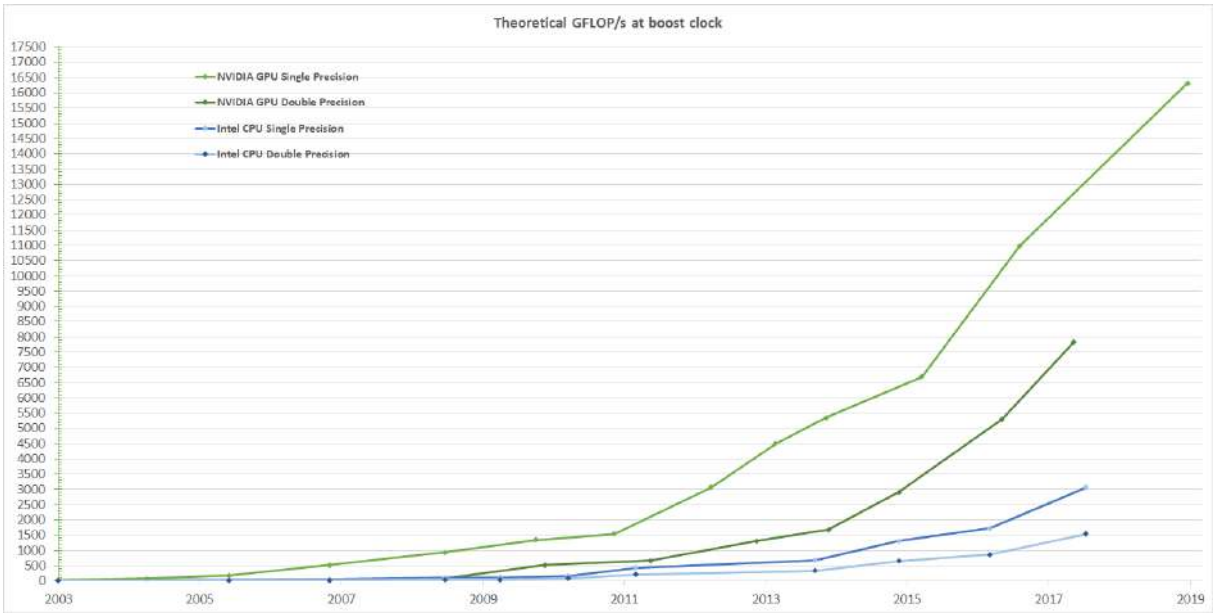
Esta seção visa apresentar os conceitos fundamentais sobre os quais o presente trabalho se sustenta e sem os quais não seria possível discorrer sobre as seções que se seguem. Primeiramente é apresentada a arquitetura das placas de vídeo da NVIDIA, seguida do princípio de funcionamento do CUDA e por fim o algoritmo utilizado na análise.

2.1 Unidades de Processamento Gráfico

As placas de vídeo são projetadas para efetuar quantidades massivas de cálculos por segundo, a fim de renderizar imagens de alta resolução sem comprometer a sua taxa de quadros, um problema inerente de *video-games*. De fato a evolução das placas gráficas foi, de início, quase que exclusivamente impulsionada pelos *video-games*, mas quando seu potencial computacional foi percebido, elas passaram a ser utilizadas para as mais diversas aplicações (Owens et al., 2008).

A crescente demanda por capacidade de processamento gráfico continua a impulsionar o desenvolvimento de placas gráficas cada vez mais poderosas, conforme demonstrado nas figuras 1 e 2 (NVIDIA, 2020).

Figura 1: Comparativo entre quantidade de operações em ponto flutuante por segundo em CPU e em GPU.



Fonte: NVIDIA (2020).

Figura 2: Comparativo entre largura de banda de memória em CPU e em GPU.

Fonte: NVIDIA (2020).

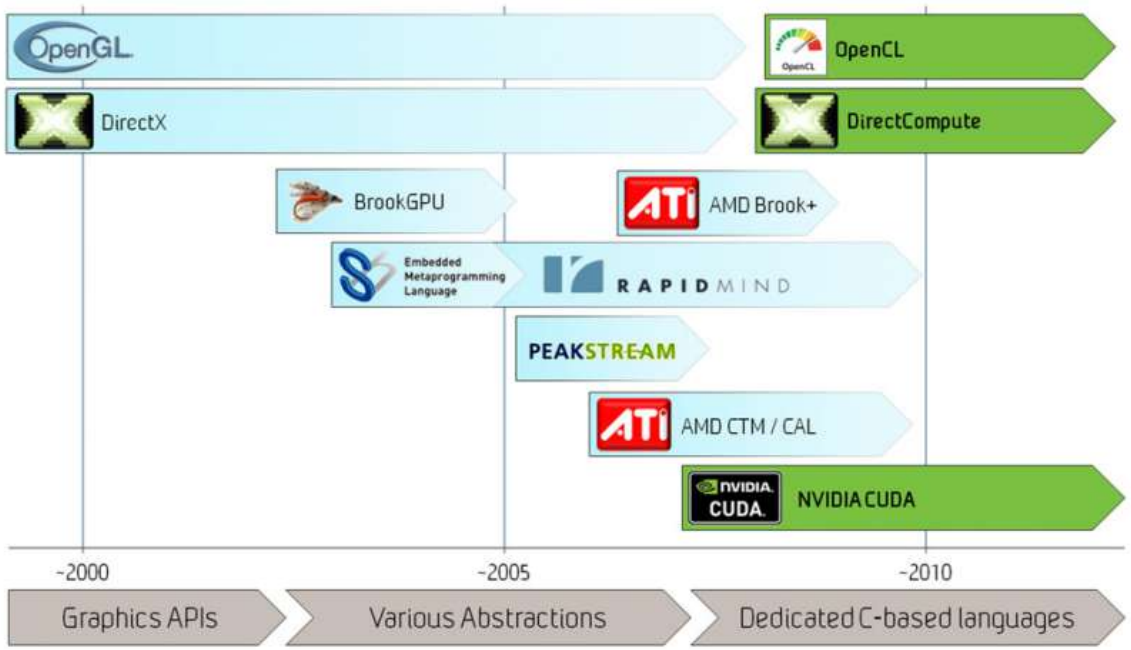
2.1.1 Histórico

Conforme seu nome sugere, as primeiras *graphics processing units* (GPUs) possuíam *hardware* dedicado a renderização gráfica, sendo concebidas para efetuar operações utilizando apenas primitivas geométricas, mais especificamente pontos, linhas e triângulos. Formas geométricas complexas por sua vez eram divididas em múltiplas faces triangulares, cujo brilho, matiz e saturação eram calculadas e direcionadas a seus respectivos *pixels* (Arora, 2012; Kim et al., 2012).

As primeiras GPUs trabalhavam com uma quantidade significativamente limitada de operações, que tinham como único propósito converter estruturas no espaço tridimensional em imagens bidimensionais (Owens et al., 2008). Era difícil desenvolver e optimizar programas voltados a GPUs, mas quando tais programas funcionavam eles demonstravam uma incrível melhora no quesito de velocidade quando comparado com códigos similares que eram executados em CPU, o que conduziu ao desenvolvimento de novas e intuitivas formas de programar em GPU (Owens et al., 2008).

Quando as GPUs começaram a ser usadas para aplicações não gráficas, tais aplicações tinham que ser reformuladas para se adequar linguagens de *shaders*, como OpenGL ou DirectX por exemplo, sendo necessário descrever problemas de todos os tipos através de primitivas gráficas apenas. A dificuldade de se fazer isso levou ao desenvolvimento de outras formas de utilizar as GPUs visando facilitar o processo (Nickolls and Dally, 2010; Kim et al., 2012).

Figura 3: Cronologia das linguagens de programação de GPUs.



Fonte: Owens et al. (2008).

O avanço no *hardware* das GPUs também foi direcionado no sentido de melhorar sua programabilidade, inclusive para aplicações de uso geral, possibilitando até mesmo o surgimento de ferramentas com CUDA, DirectCompute, and OpenCL, que atualmente são as predominantes no uso de GPGPUs (Owens et al., 2008). O CUDA *toolkit* da NVIDIA permite ao programador utilizar uma abordagem mais tradicional ao lidar com sua GPU, de forma que ele possa programar em uma linguagem já familiar a ele, como C ou Fortran, por exemplo (Kim et al., 2012). A figura 3 ilustra de forma simplificada a cronologia do desenvolvimento das linguagens de programação de GPUs.

2.2 Arquitetura Das placas gráficas da NVIDA

A evolução da arquitetura das GPUs e das CPUs tomaram direções bem diferentes. As CPUs evoluíram para executar uma única tarefa por vez, uma após a outra, sistematicamente, aplicando todos seus recursos e potencial em cada operação, de forma a tomar a menor quantidade de tempo possível em cada tarefa. As GPUs por sua vez utilizam seus recursos para executar grandes quantidades de trabalho independente entre si a cada ciclo, isso porque a resolução das imagens que deve renderizar continuam crescendo, mas a quantidade de imagens que o olho humano consegue perceber por segundo se mantém inalterada (Owens et al., 2008) Essa diferença de paradigma teve consequências claras no *hardware* dos dispositivos, conforme ilustra a figura 4.

O fluxo de trabalho das GPUs pode ser definido como do tipo *feed-forward* de múltiplas etapas, ou seja, os resultados obtidos a cada ciclo de trabalho efetuado por

uma GPU são diretamente usados no ciclo seguinte e assim sucessivamente. Ao seguir esse modelo de funcionamento todos os ciclos de trabalho demandam elevado grau de paralelismo para atuar com eficiência, e não por coincidência, o hardware das GPUs foi desenvolvido para proporcionar exatamente isso. Quando comparada com as CPUs, a execução de cada tarefa é mais lenta, mas nesse meio tempo a quantidade de trabalho executado tende a ser muito maior, tornando o processo como um todo mais rápido em diversas situações (Owens et al., 2008).

Figura 4: Comparativo entre o *hardware* das GPUs e das CPUs.

Fonte: NVIDIA (2020).

A arquitetura das GPUs da NVIDIA é formada basicamente por um conjunto de *streaming multiprocessors* (SMs), DRAM (Dynamic Random Access Memory) de alta largura de banda e registradores, conforme demonstrado na figura 5 (Arora, 2012). Os diferentes modelos de GPUs da NVIDIA variam no número de *streaming multiprocessors* (SMs) e na quantidade de memória RAM. Quanto mais numerosos são os SMs em uma GPU, melhor sua performance computacional, e quanto mais memória a GPU possui, maior sua *memory bandwidth* (Nickolls and Dally, 2010).

A comunicação entre CPU e GPU se da por meio de barramento PCIe (Peripheral Component Interconnect Express), enquanto a comunicação entre GPUs é feita pela interface NVLink. O acesso a memória DRAM da GPU por parte dos seus SMs é interfaceado por controladores de memória que trabalham de forma independente uns dos outros, permitindo acessos paralelos e direcionados, mas que mantêm alta a largura de banda (Nickolls and Dally, 2010).

Cada *thread* representa um fluxo de operações paralelo às demais *threads*, tendo seus próprios registradores, memória, pilha, endereço de instrução e flags (Nickolls and Dally, 2010). A quantidade de *threads* que compõem um *thread block* é definida pelo usuário, até um limite que pode variar a depender do modelo de GPU utilizado, como será visto mais adiante. Já as chamadas *warps* são conjuntos de 32 *threads* executadas

Figura 5: Exemplo de uma arquitetura de GPU.

Fonte: adaptada de NVIDIA (2016).

simultaneamente por um mesmo SM, o que pode ter mais ou menos *threads* que as de um *thread block* (Arora, 2012).

GigaThread é a unidade responsável por direcionar *CUDA threads* a SMs disponíveis, distribuindo o trabalho da melhor forma possível. Os SM por sua vez organizam as *threads* em blocos e as executam. Conforme cada *thread block* termina de executar suas tarefas e liberam os recursos de seu respectivo SM, a *GigaThread* atribui a esse SM novas tarefas (Nickolls and Dally, 2010).

Os SMs das GPUs modernas são compostos basicamente por unidades de processamento, unidades de gerenciamento e memória (Nickolls and Dally, 2010). Algumas dessas unidades são:

- *CUDA cores*: são os núcleos de processamento da GPU, realizam um cálculo aritmético de número inteiro ou de flutuante (32bits) a cada ciclo de trabalho;
- *Special function units*: são projetadas para rápido calculo de funções de raiz quadrada, seno, cosseno, exponencial e logarítmica usando ponto flutuante;
- *Load/store units*: executam instruções de acesso a memória. Essas unidades combinam o acesso múltiplas *threads* por vez, de forma a minimizar o número de acessos;
- *Double precision unit*: efetuam cálculos aritméticos de ponto flutuante com 64bits;

- *Warp scheduler*: seleciona, a cada ciclo de trabalho, uma *warp* para executar instruções;
- *Dispatch units*: despacham instruções retiradas da fila do *instruction cache* às *warps* que irão executá-las.

Figura 6: Exemplo de um *streaming multiprocessor*.

Fonte: adaptada de NVIDIA (2016).

Os SMs compartilham entre si memória cache de segundo nível interfaceada com a DRAM da GPU, mas cada SM também possui sua própria memória cache de primeiro nível. Cada SM é independente e dispõe de núcleos de processamento e memória o suficiente para rodar um ou mais *CUDA thread blocks*. Somando o potencial de todos os SM de uma GPU é possível rodar milhares de *threads* simultaneamente (Nickolls and Dally, 2010).

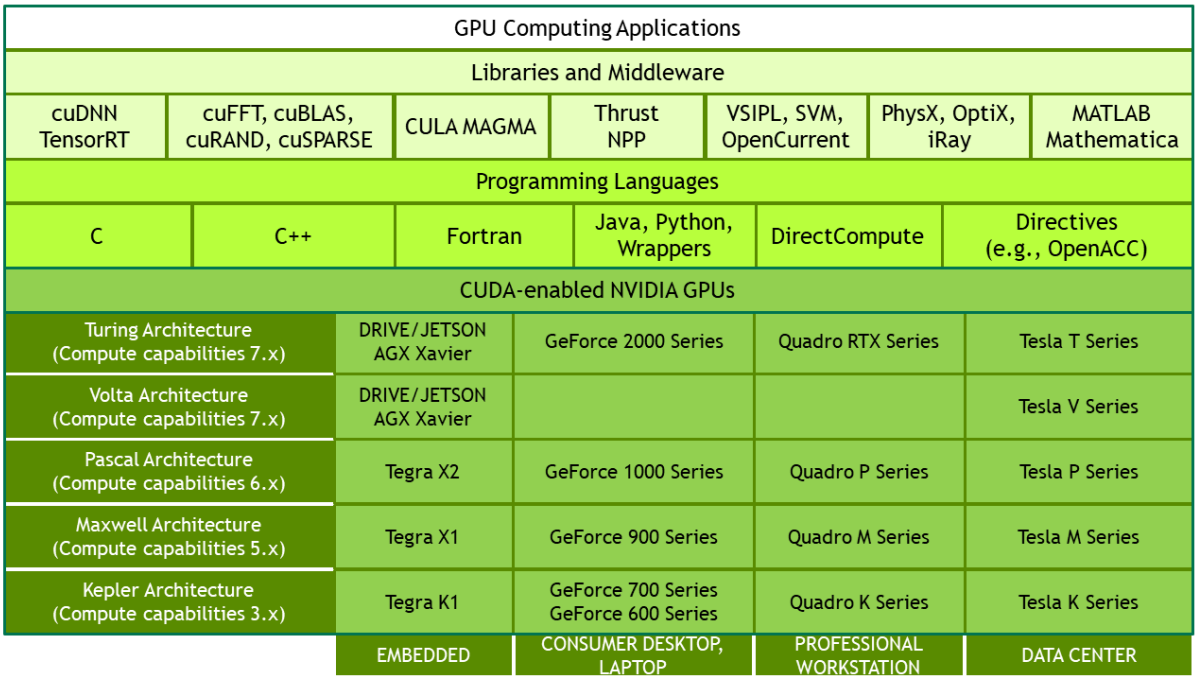
A *shared memory* contida nos SMs, como seu nome sugere, é compartilhada entre todas as *threads* em um mesmo *thread block*. O rápido acesso a *shared memory* aumenta significativamente a performance de muitas aplicações, enquanto ao mesmo tempo diminui a requisição de dados na DRAM (Nickolls and Dally, 2010).

2.3 *CUDA toolkit*

CUDA é uma plataforma de computação desenvolvida pela NVIDIA capaz de aplicar o elevado potencial de paralelização das placas de vídeo em computação de uso geral de forma simples e eficiente (NVIDIA, 2020).

Seu *toolkit* possui de um conjunto muito amplo de ferramentas para expressar paralelismo, mesmo usando de linguagens intrinsecamente seriais para isso, o que permite ao programador criar algoritmos eficientes e com elevado grau de paralelismo utilizando uma linguagem já familiar (Nickolls and Dally, 2010).

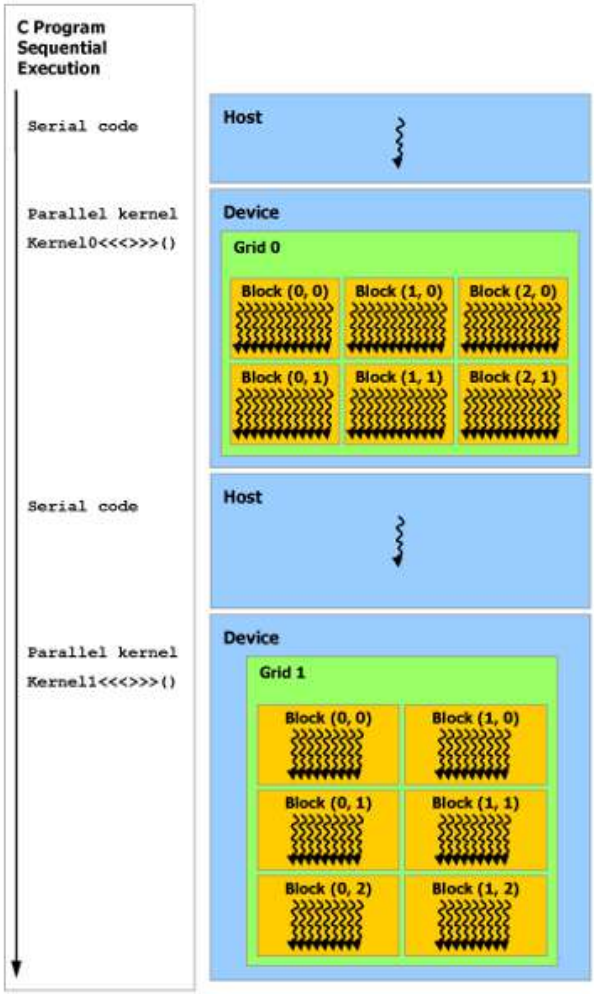
Figura 7: Linguagens de programação e aplicações suportadas pelo *CUDA toolkit*.



Fonte: NVIDIA (2020).

Programas desenvolvidos com CUDA usam o modelo de Programação Heterogênea, ou seja, uma parte deles roda sequencialmente na CPU enquanto a outra parte roda na GPU utilizando *threads* paralelas, conforme ilustrado na figura 8. A CPU é o *host*, responsável por coordenar todo o fluxo de execução em um programa que utiliza CUDA (NVIDIA, 2020). A parte sequencial desses programas opera como em qualquer linguagem serial, o diferencial aqui é a utilização da GPU, que será abordada utilizando linguagem C.

Figura 8: Fluxo de execução de um programa utilizando CUDA.



Fonte: NVIDIA (2020).

A CPU instancia os chamados *kernels*, que são funções descritas de forma serial no programa, mas que utilizam múltiplas *threads* ao ser executadas na GPU, referenciada daqui em diante como *device* (NVIDIA, 2020). Para chamar um *kernel* deve-se usar a seguinte sintaxe:

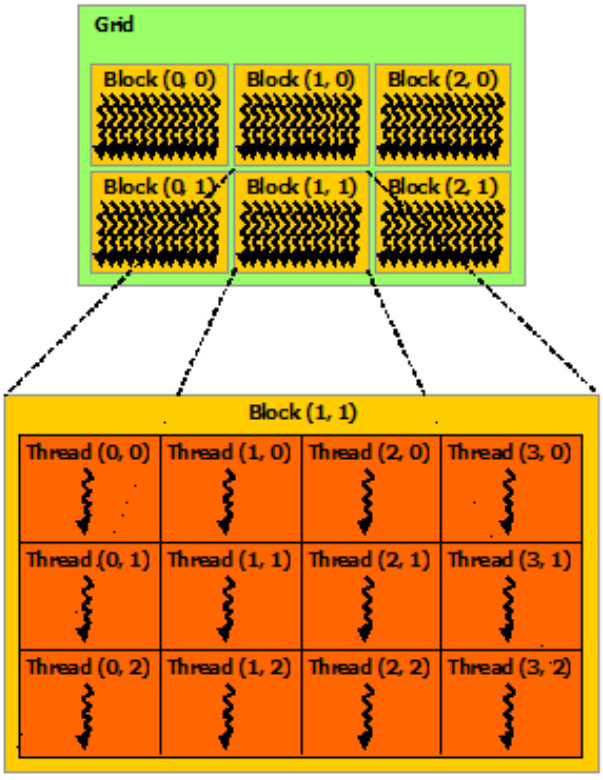
```
1  // Declaração do Kernel
2  __global__ void Nome (definição das entradas){
3      ...
4  }
5  void main(){
6      ...
7      // Chamada do Kernel
8      Nome <<< block, thread, shared, stream[i] >>> (entradas);
9      ...
10     }
```

O especificador "___global___"é usado junto com a declaração do *kernel* para diferenciá-lo de uma função normal, significando que ele é visto por ambos dispositivos. Também existem os especificadores "___host___"e "___device___", o uso do primeiro ou de nenhum tem o mesmo significado, o de uma função exclusivamente do *host*, enquanto o segundo é uma função exclusiva do *device* e que deve ser chamada por um *kernel* (NVIDIA, 2020).

Já os argumentos entre $<<<...>>>$ definem como o *device* vai trabalhar com o *kernel*. Os dois últimos argumentos serão discutidos mais adiante, já os outros dois definem, respectivamente, a quantidade de *thread blocks* em uma *grid* e a quantidade de *threads* em um *thread block*. Como tanto a *grid* quanto os *thread blocks* podem ser unidimensionais, bidimensionais ou tridimensionais, os argumentos utilizados são do tipo "dim3", estruturas de dados compostas por 3 variáveis do tipo "size_t"que indicam respectivamente a quantidade de elementos nas dimensões x, y e z (NVIDIA, 2020).

A forma como as *threads* são organizadas para gerar *thread blocks* e *grids* pode ser vista na figura 9. Cada *thread* conhece sua posição em um *thread block*, assim como cada *thread block* conhece o número de *threads* que o compõem e sua posição na *grid*, informações essas acessíveis ao usuário também por meio de variáveis do tipo "dim3", "threadIdx", "blockDim"e "blockIdx", respectivamente (NVIDIA, 2020).

Figura 9: Formação de uma *grid* de *threads* paralelas.



Fonte: NVIDIA (2020).

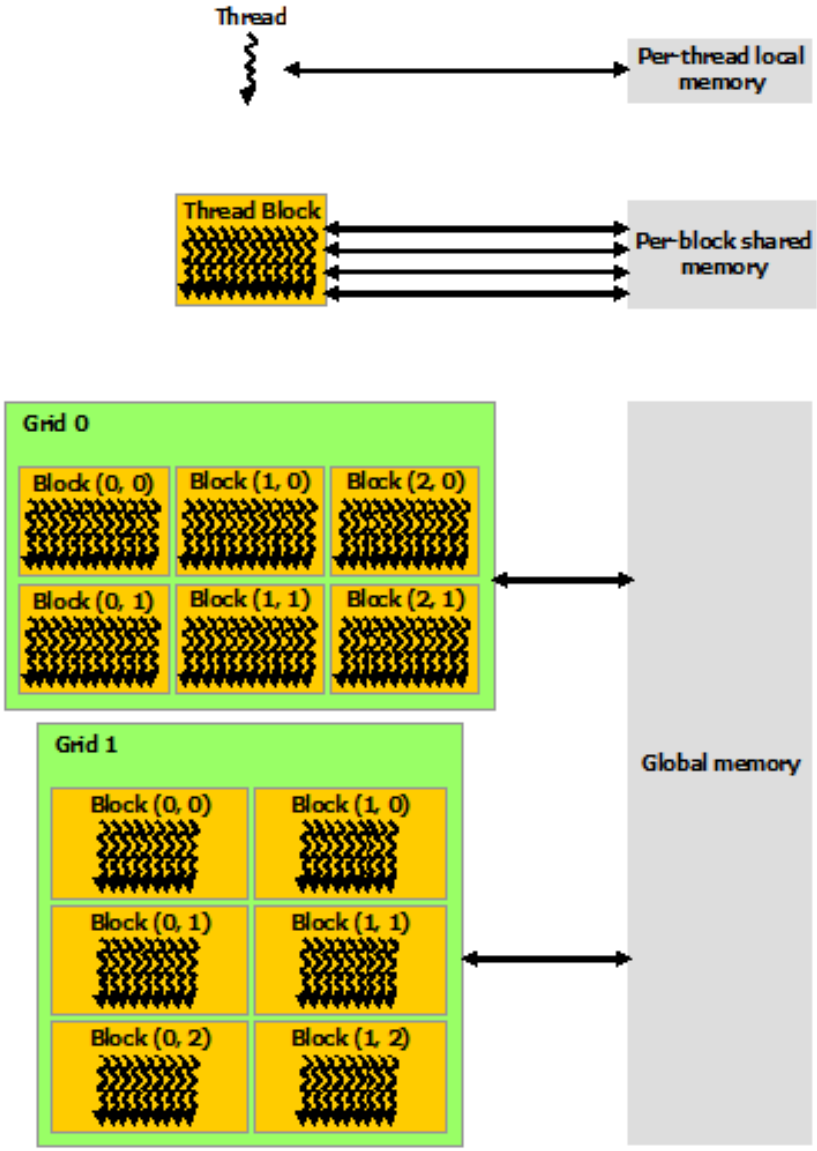
A forma como as diferentes *threads* usam recursos de memória do *device* é chamada de Hierarquia de Memória. Conforme demonstrado pela figura 10, memória local só pode ser acessada pela *thread* em que foi criada, *shared memory* só pode ser acessada por *threads* que pertençam a um mesmo *thread block* e *global memory* pode ser acessada por todas as *threads* (NVIDIA, 2020).

A função cudaMalloc é usada no *host* para alocar *global memory* dinamicamente na DRAM do *device*, enquanto a função cudaMemcpy copia dados do *host* para o *device* e vice versa. Por outro lado a memória local e a *shared memory* são declaradas nos *kernels* (NVIDIA, 2020), conforme se segue.

```
__global__ void kernel (definição das entradas){
    ...
    // Posição da thread nas dimensões x, y e z
    int x = blockDim.x * blockIdx.x + threadIdx.x;  // memória local
    int y = blockDim.y * blockIdx.y + threadIdx.y;  // memória local
    int z = blockDim.z * blockIdx.z + threadIdx.z;  // memória local

    // shared memory
    __shared__ float shared_mem[];
    ...
}
void main(){
    ...
    float* h_mem;                          // host memory
    ...
    float* d_mem;                          // global memory

    // aloca n bytes na global memory do device
    cudaMalloc((void**)& d_mem, n);

    // copia n bytes de h_mem para d_mem
    cudaMemcpy(d_mem, h_mem, n, cudaMemcpyHostToDevice);
    ...
}
```

É considerada uma boa prática diferenciar memória exclusiva do *host* (CPU) da *global memory* (GPU), pois elas só podem ser acessadas pelos dispositivos ao qual pertencem. Um padrão frequentemente adotado é utilizar o prefixo "d_"no nome de variáveis que armazenam *global memory* e o prefixo "h_"nas variáveis exclusivas do *host* (NVIDIA, 2020).

Figura 10: Hierarquia de memória.



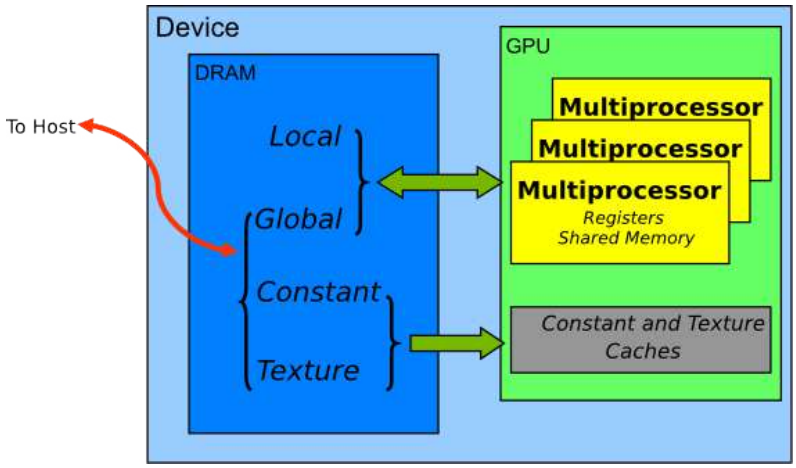
Fonte: NVIDIA (2020).

A quantidade de bytes de *shared memory* a serem alocados é o terceiro argumento entre $<<<...>>>$ mencionado anteriormente. A vantagem em se utilizar *shared memory* e memória local em relação a *global memory* é que as primeiras são armazenadas em registradores que já se encontram nos SMs e por isso os dados percorrem um trajeto menor até as *threads*, aumentando a velocidade dos acessos (NVIDIA, 2020).

Existem também outras formas de se manejar acessos a memória, que são utilizando *constant memory*, *pinned memory* ou *texture memory*. A *constant memory* é armazenada diretamente no *device*, mas diferentemente da *shared memory* ela é acessível a todas as *threads*. Por outro lado ela possui uma série de desvantagens, dentre elas a impossibilidade de ser alocada dinamicamente e possuir uma capacidade de armazenamento significativamente baixa, 64KB apenas. Além disso o acesso a *constant memory* é limitado, se muitas *threads*

tentarem acessar endereços de *constant memory* diferentes ao mesmo tempo, sua utilização fica prejudicada e o processo pode ficar mais lento do que seria com a *global memory* (NVIDIA, 2020).

Figura 11: Localização dos diferentes tipos de memória.



Fonte: NVIDIA (2020).

De fato a *constant memory* é um recurso útil em situações bem específicas apenas. O comando para alocar esse tipo de memória é feito no *host* utilizando o especificador "___constant___"e os dados endereçados nesse espaço de memória colocados lá são copiados de algum endereço de memória da *host* por meio da função cudaMemcpyToSymbol (NVIDIA, 2020), conforme se segue.

```
1  ...
2  __constant__ float c_mem[512];          // constant memory
3  ...
4  void main(){
5      ...
6      float* h_mem;                       // host memory
7      ...
8      // copia n bytes de h_mem para c_mem
9      cudaMemcpyToSymbol(c_mem, h_mem,  n);
10     ...
11 }
```

Os *Texture objects* são geralmente uma alternativa mais viável. Criados para referenciar e manejar o acesso a *texture memory*, eles são criados a partir de *cuda arrays* e estruturas de dados que contêm uma série de características a seu respeito. O *cuda arrays* por sua vez são vetores 1D, 2D ou 3D optimizados para o acesso a *texture memory* e também são criados com base em descritivos contidos em uma estrutura de dados (NVIDIA, 2020). Para maiores detalhes sobre ambas as estruturas é recomendável consultar a documentação

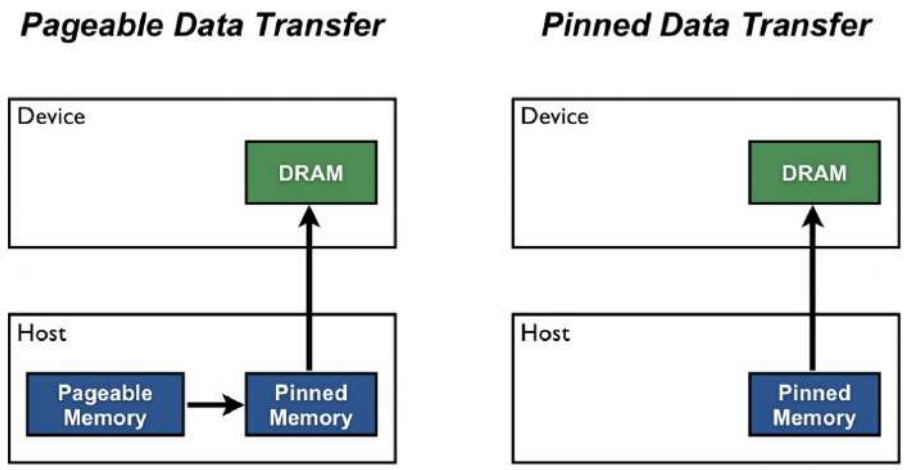
do *toolkit* no site da NVIDIA (2020), mas um breve exemplo de como utilizar *texture object* é apresentado abaixo:

```
__global__ void kernel (cudaTextureObject_t tex){ ... }

void main(){
    ...
  //Estrutura de dados que especifica o tipo de variável
  // a ser armazenado pelo cuda array, float no caso
  cudaChannelFormatDesc channelDesc =
                  cudaCreateChannelDesc<float>();
  //Declaração e alocação de memória do cuda Array
  cudaArray *d_cudaArray;
  cudaMalloc3DArray(&d_cudaArray, &channelDesc,
                    make_cudaExtent(bytes), 0);
  //Estrutura de dados que  Array
  cudaMemcpy3DParms copyParams = { 0 };
  cudaMemcpy3D(&copyParams);

  //Estruturas de dados que descrevem o texture object
  cudaResourceDesc    texRes;
  cudaTextureDesc     texDescr;

  // Efetivamente criando o texture object
    cudaTextureObject_t tex;
    cudaCreateTextureObject(&tex, &texRes, &texDescr, NULL);
    ...
}
```

Armazenada diretamente no *device*, a *texture memory* não só é projetada para permitir múltiplos acessos simultâneos a um mesmo endereço de memória de forma eficiente, como também é organizada de forma a optimizar esses acessos a endereços adjacentes. Para acessar endereços de *texture memory* é necessário utilizar as funções tex1D, tex2D ou tex3D, a depender do número de dimensões do *texture object*. A forma como tais funções fazem os acessos pode ser configuradas para acessos a endereços precisos ou até mesmo aproximar e interpolar resultados (NVIDIA, 2020).

A *pinned memory* por sua vez é um tipo de memória pertencente ao *host*, mas que possui um diferencial importante em relação a *pageable memory*, que é o tipo de memória padrão. Como a figura 12 ilustra, o *device* não pode diretamente acessar *pageable memory*, por isso todas as transferências dados entre dispositivos são feitas por intermédio de *pinned memory* (NVIDIA, 2020).

Figura 12: Funcionamento da *pinned memory*.



Fonte: NVIDIA (2020).

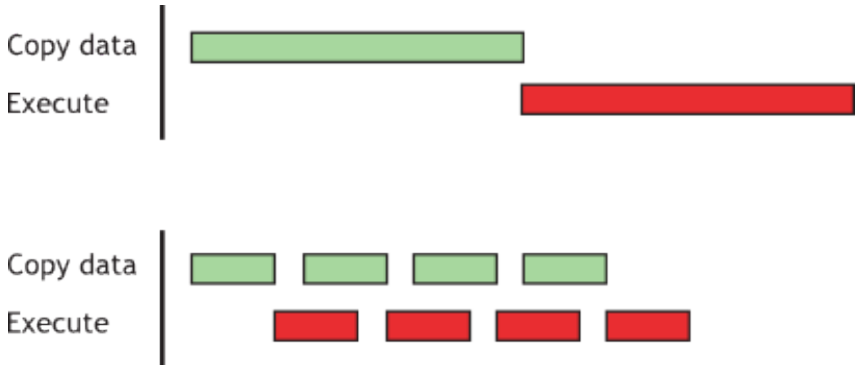
```
void main(){
    ...
    // pinned memory
    float   *h_pinned;
    ...

    // aloca n bytes de pinned memory em h_pinned
    cudaMallocHost((void**)&h_Pinned, n) );

    // copia n bytes de h_Pageable para h_Pinned
    memcpy(h_Pinned, h_Pageable, n, cudaMemcpyHostToHost);
    ...
}
```

Dados armazenados em *pinned memory* tendem a permanecer lá apenas durante as transferências, mas gravar informação diretamente em *pinned memory* tem suas vantagens, mas para isso devem ser criados *streams* extras. *Streams* são como *devices* virtuais que partilham o trabalho de um *device* físico, mas cada *stream* tem seu próprio fluxo de trabalho e atua independentemente das demais, ou seja, um mesmo *device* pode não só sobrepor tarefas de um mesmo tipo, mas também executar atividades completamente distintas concomitantemente (NVIDIA, 2020).

Quando nada é especificado todas as operações são executadas na *stream* nula, que atua de modo síncrono, sempre esperando a última tarefa ser concluída antes de iniciar a próxima. As *streams* criadas por outro lado operam de forma assíncrona, isso significa que enquanto uma *stream* transfere dados do *host* para o *device*, outra pode executar um *kernel*, optimizando o processo, conforme representado na figura 13 (NVIDIA, 2020). O trecho de código abaixo exemplifica o uso de 8 *streams* para sobrepor tarefas.

```
int main()
{
    ...
    const int num_streams = 8;
    cudaStream_t streams[nStreams]; // Declara 8 streams

    for (int i = 0; i < nStreams; i++) {
        cudaStreamCreate(&streams[i]); // Cria uma stream
        int offset = i * n / nStreams;

        // Transferência assíncrona de memória
        cudaMemcpyAsync(&d_mem[offset], &h_pinned[offset],
                        (n / nStreams + 1) * sizeof(float),
                        cudaMemcpyHostToDevice, stream[i])

        kernel<<<block, thread, shared, streams[i]>>>(entradas);

        cudaStreamDestroy(stream[i]); // Destrói uma stream
    }
    ...
}
```

Figura 13: Sobreposição de tarefas com o uso de *streams*.



Fonte: NVIDIA (2020).

Devido a natureza caótica do uso de *streams* assíncronas existem as chamadas barreiras, que são primitivas de sincronização, para garantir que as *streams* tenham terminado suas tarefas quando necessário. Quando uma *stream* termina suas tarefas e atinge uma barreira, ela permanece ociosa até que as outras *streams* também atinjam a mesma barreira, sem poder continuar (NVIDIA, 2020).

Apesar de todas as *threads* serem independentes entre si nem todas são executadas exatamente ao mesmo tempo, sendo executadas de forma ligeiramente deslocadas no

tempo, o que em algumas situações pode ser um problema. Para contornar esse problema existem também barreiras de sincronização que atuam apenas em *threads* de um mesmo *kernel*, bem como as *atomic functions* (NVIDIA, 2020).

As barreiras costumam ser suficientes quando todas as *threads* escrevem em endereços de memória diferentes, mesmo que uma *thread* modifique um endereço lido por outra, isso porque todas as leituras de memória podem ser efetuadas antes da escrita (NVIDIA, 2020), conforme abaixo.

```
__global__ void kernel (float* array)
{
   int  x = blockDim.x * blockIdx.x + threadIdx.x ;
   float tmp = array[x];
   __syncthreads();      // barreira
   array[x+1] = tmp;
    ...
}
```

Por outro lado quando muitas *threads* modificam um mesmo endereço de memória pode-se usar *atomic functions*. Esse tipo especial de função é projetado especificamente para efetuar cálculos de um mesmo tipo envolvendo muitas variáveis, como num somatório por exemplo, conforme pode ser visto abaixo (NVIDIA, 2020).

```
__global__ void somatorio (float* array, float sum)
{
   int  x = blockDim.x * blockIdx.x + threadIdx.x ;
   atomicAdd(&sum, array[x]);
}
```

Apesar de serem muito úteis nas situações para as quais foram projetadas, as *atomic functions* possuem uma baixa variedade de operações que podem ser feitas, além de limitarem a velocidade dos *kernels* por terem que sincronizar acessos de memória e por esse motivo devem ser usadas apenas quando necessário (NVIDIA, 2020).

Existem ainda muitas outras ferramentas disponibilizadas pelo CUDA *toolkit* que não foram abordadas aqui pois não serão abordadas neste trabalho. Um maior detalhamento sobre as ferramentas apresentadas aqui, bem como outras funcionalidades, pode ser encontrado na documentação do CUDA *toolkit* através do url: `https://docs.nvidia.com/cuda/`.

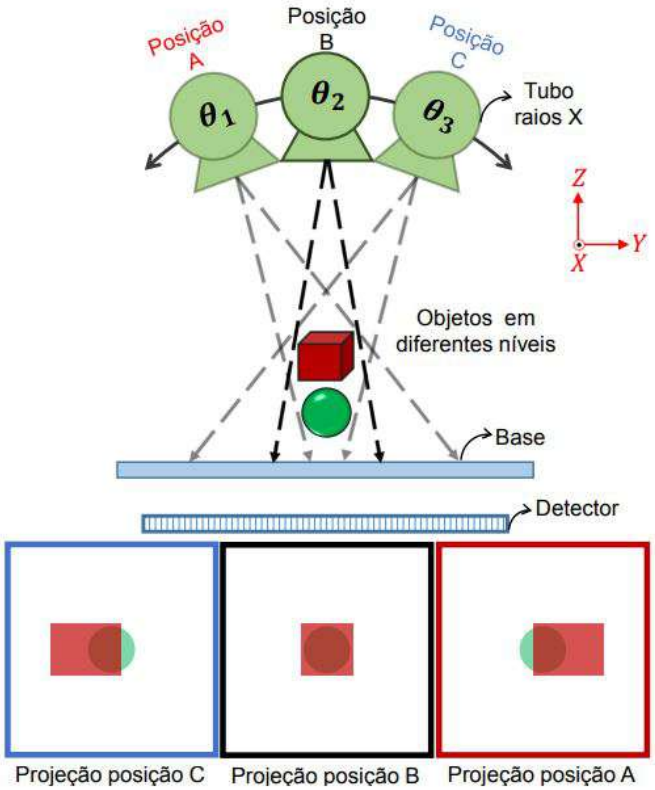
2.4 Reconstrução de Imagens para Tomossíntese Digital Mamária

Câncer é a denominação dada a um conjunto de doenças cuja principal característica é o aumento descontrolado do número de células no tecido que a enfermidade ataca, podendo inclusive criar tumores e se espalhar para outras regiões do corpo. Dentre os diversos tipos existentes, o câncer de mama foi aquele com a maior taxa de mortalidade entre mulheres no ano de 2017 no Brasil e estima-se que até o final de 2020 serão diagnosticados mais de 66 mil casos da doença (INCA, 2020).

A detecção do câncer de mama em sua etapa inicial possibilita que sejam feitos tratamentos menos invasivos e aumenta a taxa de recuperação dos pacientes, por isso seu diagnóstico é tão importante e é recomendado que mulheres entre 50 e 69 anos façam exames de rastreamento a cada dois anos, mesmo na ausência de sintomas (INCA, 2020).

O exame mais conhecido que atende a essa finalidade é a mamografia, que consiste da obtenção de uma simples imagem radiográfica da mama, e como toda imagem radiográfica, possui limitações. Uma radiografia é a representação bidimensional de uma estrutura tridimensional, e isso por si só implica em perda de informação, tornando possível que estruturas se sobreponham umas as outras e impossibilite um diagnóstico preciso (Vedantham S. and et al, 2015; ASC, 2020).

Figura 14: Aquisição das projeções por um equipamento de tomossíntese.



Fonte: Vimieiro, Rodrigo de Barros (2019).

Para contornar esse problema foi desenvolvida a tomossíntese digital mamária, uma técnica que visa reconstruir o volume 3D do objeto de análise a partir da captura de múltiplas imagens radiográficas (projeções). Para obter as imagens necessárias ao algoritmo de reconstrução o tubo emissor de raios X varia levemente sua posição em relação ao objeto seguindo a trajetória de um arco, capturando várias projeções 2D do objeto tridimensional em diferentes pontos dessa trajetória (Vimieiro R. B. et al., 2019).

Como pode-se observar na figura 14, o objeto verde foi completamente obstruído pelo objeto vermelho na projeção "B", mas é parcialmente visível nas projeções "A"e "C"devido a variação angular do emissor de raios X. Depois de adquiridas as projeções é possível aplicá-las em um algoritmo de reconstrução capaz de inferir a geometria dos objetos capturados (Vimieiro, Rodrigo de Barros, 2019).

O Laboratório de Visão Computacional (LAVI), da Escola de Engenharia de São Carlos, Universidade de São Paulo, desenvolveu uma *toolbox* de reconstrução de imagens para tomossíntese digital mamária utilizando a linguagem do MATLAB, *toolbox* essa que pode executar todas as etapas compreendidas por um exame de tomossíntese digital mamária, sendo capaz até mesmo de efetuar a aquisição de projeções ao utilizar um *phantom* virtual já composto por diversas fatias como objeto de estudo. Além disso a *toolbox* também permite alterar a geometria do sistema para corresponder a qualquer equipamento de tomossíntese no mercado, assim como selecionar o método a ser utilizado para a reconstrução.

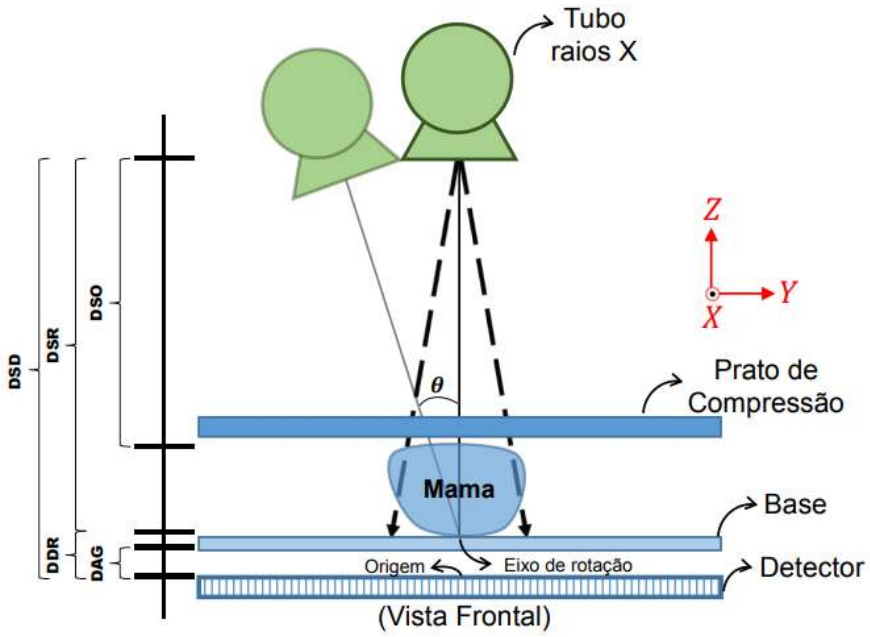
A *toolbox* está disponível no GitHub do LAVI pelo url: `https://github.com/LAVI-USP/DBT-Reconstruction` e seu processo de desenvolvimento, bem como os testes efetuados para sua validação, podem ser encontrados em detalhes nos trabalhos de Vimieiro et al. (2019) e Vimieiro, Rodrigo de Barros (2019).

Como o presente trabalho se desenvolve a partir da aplicação de uma das funcionalidades da *toolbox*, mais especificamente o algoritmo que cria as projeções, é importante estudar algumas de suas características mais atentamente.

2.4.1 Geometria do Equipamento

A *toolbox* consegue se adaptar a geometria de qualquer equipamento de tomossíntese mamária, sendo ele um equipamento já consolidado no mercado ou não, e isso se deve a uma simples estrutura de dados que contem todas as características de um equipamento de tomossíntese, inclusive características do processo de funcionamento, como por exemplo o número de projeções adquiridas e o número de fatias que o objeto reconstruído deve possuir.

Figura 15: Geometria de um equipamento de tomossíntese digital mamária.



Fonte: Vimieiro R. B. et al. (2019).

A figura 15 ilustra simplificadamente como é a geometria de um equipamento utilizado em um exame de tomossíntese digital mamária, enquanto a tabela 1 compreende os principais parâmetros contidos na estrutura de dados que descreve o sistema.

Tabela 1: Principais parâmetros que descrevem o sistema.

Parâmetro	Significado
nx	Número de voxels no eixo X (colunas)
ny	Número de voxels no eixo Y (linhas)
nz	Número de fatias
nu	Número de detectores no eixo X (colunas)
nv	Número de detectores no eixo Y (linhas)
dx	Tamanho real de um único voxel no eixo X (mm)
dy	Tamanho real de um único voxel no eixo Y (mm)
dz	Tamanho real de um único voxel no eixo Z (mm)
du	Tamanho real de um único detector no eixo X (mm)
dv	Tamanho real de um único detector no eixo Y (mm)
DSD	Distância do emissor de raios X até o plano do detector (mm)
DSO	Distância do emissor de raios X até o topo do objeto (mm)
DDR	Distância do detector até o eixo de rotação do emissor de raios X (mm)
DSR	Distância do emissor de raios X até seu eixo de rotação (mm)
DAG	Espaço entre o detector e a mesa que apoia o objeto (mm)
nProj	Número de projeções
tubeAngle	Variação angular total do emissor de raios X (graus)
tubeDeg	Angulação do emissor de raios X em cada projeção (graus)
detAngle	Variação angular total do detector (graus)
detectorDeg	Angulação do detector em cada projeção (graus)

Fonte: adaptada de Vimieiro, Rodrigo de Barros (2019).

2.4.2 Algoritmos

Outra característica importante da *toolbox* é o conjunto de algoritmos que o compõem, em especial os algoritmos de projeção e retroprojeção. Um operador de projeção geométrico, de maneira geral, é aquele que representa um objeto de N dimensões em um espaço de N - 1 dimensões, no caso o algoritmo utilizado projeta um objeto 3D no plano bidimensional de um detector, conforme a figura 14. A retroprojeção por sua vez faz o oposto da projeção, buscando reconstruir o objeto original, fatia por fatia, com base em suas projeções (Vimieiro, Rodrigo de Barros, 2019).

O método mais simples de reconstrução é aquele que simplesmente utiliza do algoritmo de projeção seguido do algoritmo de retroprojeção, sem se preocupar com o ruído gerado nesse processo, mas existem variantes desse método que buscam aprimora-lo, gerando resultados mais precisos e confiáveis, alguns dos quais também estão disponíveis para uso na *toolbox*. Os métodos de reconstrução iterativos tem se mostrado promissores, mas demandam elevado custo computacional, pois executam projeção e retroprojeção repetidas vezes em busca da convergência dos resultados (Vimieiro, Rodrigo de Barros, 2019), o que os tornam perfeitos candidatos para optimizar com a utilização de CUDA.

O presente trabalho, porém, não aborda a retroprojeção, visto que seu objetivo é meramente investigativo e não almeja necessariamente resultados, mas sim estudar as ferramentas de que o *toolkit* da NVIDIA dispõe. Por esse motivo apenas o algoritmo de projeção será estudado mais a fundo, uma vez que é a mais simples das opções. Após fazer os devidos testes e descobrir quais abordagens funcionam e quais não funcionam, fica aqui, para quem interessar, a sugestão de aplicar esse aprendizado para acelerar a execução do método iterativo de reconstrução.

Algoritmo 1: Projeção

```
Entrada: Volume3D, Parâmetros
Saída: Projeções
Início
    para cada Projeção ∈ Projeções faça
        θ ← Ângulo da Projeção;
        para cada Fatia ∈ Volume3D faça
            Calcular Y_i e X_i ∀ (y, x) ∈ Projeção ;
            Calcular i e j ∀ (Y_i, X_i);
            Projeção ← Projeção + Interpolação( Fatia, (i, j) );
        fim
    fim
Fim
```

Fonte: adaptada de Vimieiro, Rodrigo de Barros (2019).

O pseudocódigo acima ilustra como é feito o processo de aquisição das projeções por parte da *toolbox*, para cada posição assumida pelo emissor de raios X, o algoritmo percorre todas as fatias do objeto tridimensional (Volume3D) aplicando o método *Pixel Driven* (Vimieiro, Rodrigo de Barros, 2019).

Esse método por sua vez calcula as coordenadas (Y_i, X_i) que as projeções 2D de cada fatia ocupam no detector e varre todos os *voxels* dessas fatias projetando seu centro no detector com o uso das equações 2.1 e 2.2 (Vimieiro R. B. et al., 2019). Ambos os equacionamentos podem ser deduzidos a partir de simples semelhança de triângulo com base no sistema apresentado na figura 15.

$$Y_i(\theta, Y, Z) = Y + \frac{Z(DSR.sen(\theta) + Y)}{DSR.cos(\theta) + DDR - Z} \tag{2.1}$$

$$X_i(\theta, X, Z) = \frac{X(DSR.cos(\theta) + DDR)}{DSR.cos(\theta) + DDR - Z} \tag{2.2}$$

As equações 2.3 e 2.4 por sua vez convertem as coordenadas da imagem (y0, x0) em *pixels* (i, j) para que, por fim, o valor dos *voxels* projetados possam ser repartidos entre os *pixels* correspondentes por meio de interpolação linear.

$$i = \frac{Y_i}{dy} + y0 \tag{2.3}$$

$$j = -\frac{X_i}{dx} + x0 \tag{2.4}$$

3 MATERIAIS E MÉTODOS

Esta seção aborda o *software* MATLAB e as ferramentas de *phantoms 3D* utilizadas durante o desenvolvimento do presente trabalho, além de detalhar os procedimentos tomados para sua execução. De modo geral o *hardware* empregado na execução deste trabalho é descrito na tabela 2, bem como as verões de *software* utilizados.

Tabela 2: Componentes de *hardware* e versões de *software* utilizados.

Processador	Intel Core i5 - 6600k
Placa de vídeo	GeForce GTX 1060 6GB
Memória RAM	16 GB
Sistema operacional	Windows 7
MATLAB	R2018a
CUDA toolkit	9.0

3.1 *Phantoms 3D*

Phantoms são objetos de estudo desenvolvidos especialmente para simular uma estrutura tridimensional ou bidimensional composta por diferentes tecidos da anatomia humada, com o objetivo de permitir estudos consistentes a respeito de novas metodologias para exames sem a necessidade que testes sejam feitos diretamente em pacientes.

Figura 16: Representação dos *phantoms* utilizados.

(a) *Phantom* virtual de Shepp-Logan

Fonte: Shepp and Logan (1974).

(b) *Phantom* físico BR3D

Fonte: CIRS (2013).

O presente trabalho utilizou em seus testes os mesmos *phantoms 3D* empregados na validação da *toolbox* desenvolvida pelo LAVI. O primeiro é uma versão do *phantom* virtual de Shepp and Logan (1974) adaptada por SCHABEL (2006) para gerar um objeto virtual 3D com a quantidade de fatias que forem necessárias, enquanto o segundo é o *phantom 3D* físico modelo BR3D criado pela CIRS (2013). Por se tratar de um *phantom* físico, o uso do *phantom* BR3D se deu através de fatias já reconstruídas.

Como pode ser visto na tabela 3, que contém as especificações do sistema usado nos testes de cada *phantom*, o *phantom* Shepp-Logan é consideravelmente menor que o BR3D, isso é importante para comparar a eficiência do uso de CUDA em situações que exigem mais e menos processamento.

Tabela 3: Parâmetros que descrevem os sistemas de cada *phantom*.

Parâmetro	Shepp-Logan	BR3D
nx	128	1058
ny	128	1978
nz	128	107
nu	280	2394
nv	350	3062
dx	1mm	0,1mm
dy	1mm	0,1mm
dz	10mm	0,5mm
du	1mm	0,1mm
dv	1mm	0,1mm
DSD	6600mm	660mm
DSO	5100mm	580,5mm
DDR	400mm	40mm
DSR	6200mm	620mm
DAG	220mm	22mm
nProj	9	9
tubeAngle	$2,5^o$	25^o
detAngle	0^o	0^o

Fonte: *toolbox* (Vimieiro, Rodrigo de Barros, 2019).

3.2 MATLAB

MATLAB é um *software* voltado a programação baseada em matrizes com uma vasta gama de funcionalidades. Como o desenvolvimento da *toolbox*, cujo algoritmo de projeção é o objeto de estudo aqui, foi feito em MATLAB, é esperado que o presente trabalho também faça uso de MATLAB.

Dentre todas as funcionalidades do MATLAB, as que devem ser destacadas são as que fazem uso do potencial de computação paralela das GPUs. Abaixo segue uma breve explicação sobre as funções usadas neste trabalho, mas uma descrição mais detalhada pode ser encontrada na documentação do MATLAB, no site da MathWorks (2020).

A função "gpuDevice" deve ser a primeira a ser utilizada, pois é responsável por informar ao usuário se o MATLAB reconhece alguma placa de vídeo e, se sim, também disponibiliza suas características, informações essenciais para este trabalho. A tabela 4 contém o resultado gerado por essa função no computador usado no desenvolvimento deste trabalho.

Tabela 4: Resultado da função "gpuDevice".

Parâmetro	Descrição
Name	'GeForce GTX 1060 6GB'
Index	1
ComputeCapability	'6.1'
SupportsDouble	1
DriverVersion	11
ToolkitVersion	9
MaxThreadsPerBlock	1024
MaxShmemPerBlock	49152
MaxThreadBlockSize	[1024 1024 64]
MaxGridSize	[2.1475e+09 65535 65535]
SIMDWidth	32
TotalMemory	6.4425e+09
AvailableMemory	5.4431e+09
MultiprocessorCount	10
ClockRateKHz	1809500
ComputeMode	'Default'
GPUOverlapsTransfers	1
KernelExecutionTimeout	1
CanMapHostMemory	1
DeviceSupported	1
DeviceSelected	1

A função "gpuArray" por sua vez cria objetos chamados de *gpuArray*, que são armazenados na memória da GPU em vez da CPU, enquanto a função "gather" faz o processo inverso. Quando o MATLAB usa esse tipo de variável para efetuar certas operações, ele naturalmente as executa na GPU, contanto que tais operações tenham suporte para isso. Porém outra forma de se usar esses objetos é como entrada da função "arrayfun", conforme se segue:

```
1 ...
2 d_input = gpuArray( h_input ); % salva h_input na GPU
3  % executa o arquivo projection.m e salva o resultado em d_output
4 d_output =  arrayfun(@projection, d_input);
5 h_output = gather( d_output ); % salva d_output na CPU
6 ...
```

Essa função interpreta um algoritmo escrito em MATLAB, no caso o arquivo "projection.m", e o executa em múltiplas *threads*, uma para cada conjunto de elementos dos vetores de entrada instanciados pelas mesmas coordenadas. Se por exemplo um "arrayfun"receber como entrada a função F e os vetores A e B, então ela executará F(A(1),B(1)) em uma *thread*, F(A(2),B(2)) em outra e assim sucessivamente. Se o número de elementos de A e B forem diferentes a operação não irá funcionar adequadamente. Vale ressaltar que as variáveis de entrada e de saídas dessa função devem ser do tipo *gpuArray*.

Outra funcionalidade do MATLAB é função "mex", cuja finalidade é chamar funções em C, C++ ou Fortran como se fossem funções do próprio MATLAB, contanto que esses arquivos estejam escritos com a sintaxe adequada.

```
...
% compila o código projection.cpp
mex projection.cpp

% executa o código projection.cpp
output = projection(input);
...
```

A função "mex" compila o arquivo de entrada, no caso "projection.cpp", e cria um arquivo *MEX* de extensão ".mexw64" no diretório de trabalho, o qual é interpretado e executado pelo MATLAB como qualquer outra função. Os arquivos C e C++ destinados a serem usados desta forma devem substituir sua função "main"pela "mexFunction", cuja sintaxe é apresentada abaixo.

```
#include "mex.h"
void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
                 const mxArray *prhs[])
{
...
}
```

A função "mexFunction" é inserida pela biblioteca "mex.h" e faz o trabalho de interfacear os argumentos de entrada e saída do arquivo com os providenciados pelo MATLAB. A tabela 5 descreve o significado de cada um de seus argumentos.

Tabela 5: Argumentos de uma "mexFunction".

Parâmetro	Descrição
prhs	Vetor de argumentos de entrada
plhs	Vetor de argumentos de saída
nrhs	Número de argumentos ou de elementos do vetor de entrada
nlhs	Número de argumentos ou de elementos do vetor de saída

Fonte: adaptada de MathWorks (2020).

A função "mex"de fato funciona muito bem nos tipos de arquivos para os quais foi projetada, mas a extensão utilizada pelo *CUDA toolkit* é ".cu", mesmo que a linguagem de programação empregada seja C ou C++. Para contornar esse problema foi desenvolvida a função "mexcuda", que tem suporte para CUDA. A sintaxe da função "mexcuda" no MATLAB é a mesma que da "mex", mas difere no arquivo a ser compilado, como se segue:

```
...
#include "mex.h"
#include "gpu/mxGPUArray.h"
void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
                 const mxArray *prhs[])
{
    // Inicializa a API de GPU do MATLAB
    mxInitGPU();
...
}
```

Para usar a função "mexcuda", contudo, é necessário garantir que o MATLAB tenha acesso às bibliotecas do *CUDA toolkit*, cuja versão deve ser compatível com a do MATLAB, de a cordo com a tabela fornecida em sua documentação (MathWorks, 2020), mas que também pode ser acessada diretamente pelo url: `https://www.mathworks.com/help/parallel-computing/gpu-support-by-release.html`.

A linha de código abaixo atribui à variável "MW_NVCC_PATH"o diretório do compilador NVCC, garantindo que o MATLAB o reconheça e possa fazer uso do *CUDA toolkit* corretamente.

```
...
setenv('MW_NVCC_PATH','CUDA_dir\CUDA\v9.0\bin');
...
```

3.3 Metodologia

Este trabalho visa investigar o potencial computacional das GPUs utilizando como objeto de estudo uma *toolbox* desenvolvida em MATLAB, então é natural seguir uma metodologia que também aborde as funcionalidades do MATLAB. A metodologia empregada aqui foi a mesma apresentada na página "Illustrating Three Approaches to GPU Computing: The Mandelbrot Set" da documentação do MATLAB, acessível pelo url: `https://www.mathworks.com/help/parallel-computing/examples/illustrating-three-approaches-to-gpu-computing-the-mandelbrot-set.html`.

O método consiste em comparar o tempo de execução entre as diferentes abordagens de uso da GPU pelo *software* ao resolver um mesmo problema. Para isso foram empregadas as funções "gpuArray", "arrayfun" e "mexcuda", descritas na seção acima, na execução do algoritmo de projeção.

A título de validar os resultados obtidos por cada uma das abordagens, o código original foi utilizado como controle, ou seja, todas as projeções geradas por outros métodos foram comparadas com as originais. O critério utilizado para a validação foi aplicar as métricas de comparação de imagem "Pico da Relação Sinal Ruído" (PSNR - Peak Signal to Noise Ratio) e o "Índice de Similaridade Estrutural" (SSIM - *Structural Similarity Index*).

A métrica PSNR interpreta as diferenças de intensidade entre cada *pixel* das imagens degradadas e da imagem original e entrega um valor em dB que expressa o quão pura é a imagem testada. Alguns tipos de degradação, porém, podem influenciar mais a visualização de imagens do que outros, mesmo que estejam presentes em menor intensidade, e por isso também foi utilizada a métrica SSIM, que foi projetada especificamente para avaliar a qualidade visual de imagens. A SSIM entrega um valor entre -1 e 1, sendo que -1 significa que as imagens não possuem similaridade, enquanto 1 indica que são elas idênticas (Zhou Wang et al., 2004).

Para aplicar *gpuArray* bastou usar variáveis salvas na memória da GPU como entrada do algoritmo de projeção, mas para usar a função "mexcuda"o código originalmente escrito em MATLAB foi completamente traduzido para C++, o que de maneira geral foi uma simples adaptação de sintaxe, uma vez que a maioria das primitivas e funções utilizadas são comuns a ambas as linguagens. Entretanto a função "interp2" fugiu a essa regra e teve de ser escrita manualmente em C++. Tal função, da forma que foi empregada, interpola linearmente os *pixels* de uma imagem em suas coordenadas correspondentes no detector, que foram calculadas previamente. A figura 17 representa visualmente o funcionamento de uma interpolação bilinear, cujo equacionamento presente em 3.1 serviu de base para o desenvolvimento do algoritmo 2, que foi escrito em C++.

Após constatado que o código de projeção feito em C++ funcionou, foi hora de adaptá-lo de forma a utilizar múltiplas *threads* com CUDA. Para isso foram criados *kernels* para cada etapa que necessitaria de um número distinto de *threads*, sendo elas o cálculo de Y, o cálculo de X e os cálculos de interpolação. O uso da função "arrayfun" no MATLAB foi similar a esta etapa em específico, mas se limitou aos cálculos de X_i e Y_i, uma vez que a função de interpolação exige acesso a mais de um pixel por vez, o que a função "arrayfun" não foi projetada para fazer.

Algoritmo 2: Interpolação

Entrada: Fatia, X_i, Y_i

Saída: Interpolação

Início

para *cada* $j \in X_i$ **faça**

x1 ← a parte inteira de j;

x2 ← x1 + 1;

alfa1 ← a parte fracionária de j;

alfa2 ← 1 - alfa1;

para *cada* $i \in Y_i$ **faça**

y1 ← a parte inteira de i;

y2 ← y1 + 1;

beta1 ← a parte fracionária de i;

beta2 ← 1 - beta1;

se *(y1,x1)* ∈ *Fatia* **então**

Q11 ← Fatia(y1,x1);

senão

Q11 ← 0;

fim

se *(y1,x2)* ∈ *Fatia* **então**

Q21 ← Fatia(y1,x2);

senão

Q21 ← 0;

fim

se *(y2,x1)* ∈ *Fatia* **então**

Q12 ← Fatia(y2,x1);

senão

Q12 ← 0;

fim

se *(y2,x2)* ∈ *Fatia* **então**

Q22 ← Fatia(y2,x2);

senão

Q22 ← 0;

fim

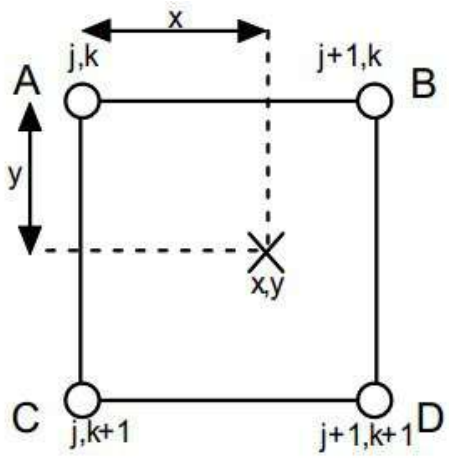
Interpolação ← Interpolação + (Q11 * alfa2*beta2) +
(Q21 * alfa1*beta2) + (Q12 * alfa2*beta1) + (Q22 * alfa1*beta1);

fim

fim

Fim

Figura 17: Interpolação Bilinear.



Fonte: K.T. Gribbon et al. (2003).

$$X_{x,y} = (1-x)(1-y)A + x(1-y)B + (1-x)yC + xyD \tag{3.1}$$

Aqui é importante ressaltar que o uso da *atomic function* "atomicAdd"foi necessário no *kernel* responsável pela execução do algoritmo de interpolação, uma vez que muitas *threads* simultâneas têm que escrever no mesmo endereço de memória.

Depois de testadas as modificações mencionadas acima, foram também feitas outras adaptações do sentido de aumentar o nível de paralelismo dos algoritmos em CUDA. Um código foi adaptado para lançar em cada operação uma quantidade de *threads* suficiente para executar os cálculos de todas as fatias simultaneamente, dispensando o uso de um *loop* executado na CPU, enquanto outro código lançou em cada operação uma quantidade de *threads* ainda maior, a fim de executar os cálculos de todas as fatias de todas as projeções simultaneamente, dispensando o uso de ambos os *loops*.

Dentre todas as versões de código criadas, algumas foram escolhidas para se aplicar ferramentas mais pontuais de CUDA, tais quais *shared memory*, *texture memory* e *streams*. O uso de *shared memory* não exigiu mudanças significativas no código, mas como a vantagem proposta no uso de múltiplas *streams* era executar trechos do algoritmo simultaneamente com transferência de dados entre dispositivos, foi necessário inverter a posição dos *loops* das fatias com o das projeções, conforme demonstrado no algoritmo 3.

A adaptação foi feita pois a obtenção da primeira projeção já necessitava de acesso a todos os dados que seriam transferidos para a GPU durante a execução de todo o algoritmo, desperdiçando grande parte do potencial de sobreposição disponibilizado pela ferramenta. Ao alterar o algoritmo dessa forma, as fatias podem ser copiadas na memória da GPU em etapas e, teoricamente, executar o código mais rapidamente.

Algoritmo 3: Projeção Adaptada

Entrada: Volume3D, Parâmetros

Saída: Projeções

Início

para *cada Fatia* $\in$ *Volume3D* **faça**

para *cada Projeção* $\in$ *Projeções* **faça**

$\theta \leftarrow$ Ângulo da Projeção;

Calcular Y_i e X_i $\forall$ (y, x) $\in$ Projeção ;

Calcular i e j $\forall$ (Y_i, X_i);

Projeção $\leftarrow$ Projeção + Interpolação(Fatia, (i, j));

fim

fim

Fim

O uso de *texture memory* por sua vez exigiu a aplicação de várias outras ferramentas e de uma série de descritivos contidos em estruturas de dados, mas abriu a possibilidade para estudos interessantes, como a possibilidade de se efetuar a interpolação sem a necessidade de aplicar o algoritmo 2.

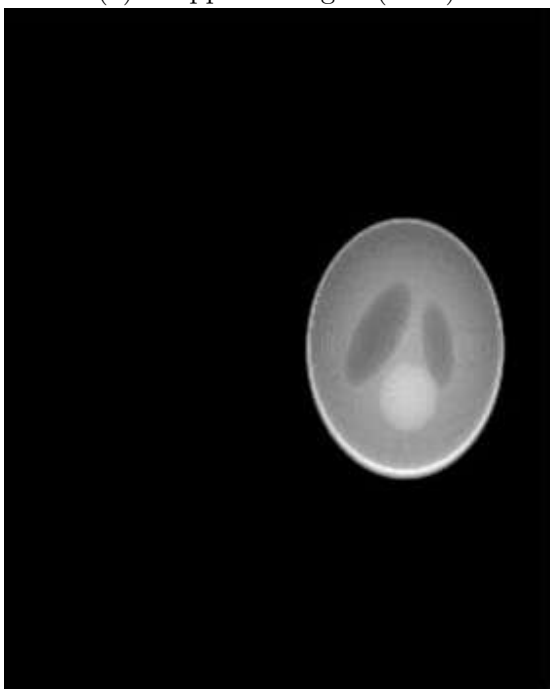
O modelo da placa de vídeo faz bastante diferença no uso de muitas das ferramentas do *CUDA toolkit*, como por exemplo a quantidade de *streams* que a GPU consegue lançar. Por esse motivo todas as informações pertinentes foram pesquisadas nas tabelas "Feature Support per Compute Capability"e "Technical Specifications per Compute Capability"presentes na documentação do *CUDA toolkit* (NVIDIA, 2020), não apenas garantindo que o modelo de placa gráfica empregada no desenvolvimento deste trabalho conseguiria utilizar os recursos pretendidos, mas também que sua execução fosse feita dentro dos limites para os quais foi projetada.

4 RESULTADOS E DISCUSSÕES

Esta seção apresenta os resultados obtidos seguindo a metodologia supracitada de forma ordenada, discorrendo brevemente sobre cada um deles. A figura 18 ilustra duas das projeções geradas pelo algoritmo original, enquanto a tabela 6 apresenta o tempo (T_c) levado para a obtenção das projeções.

Figura 18: Projeções obtidas com o algoritmo de controle.

(a) Shepp and Logan (1974)



(b) BR3D

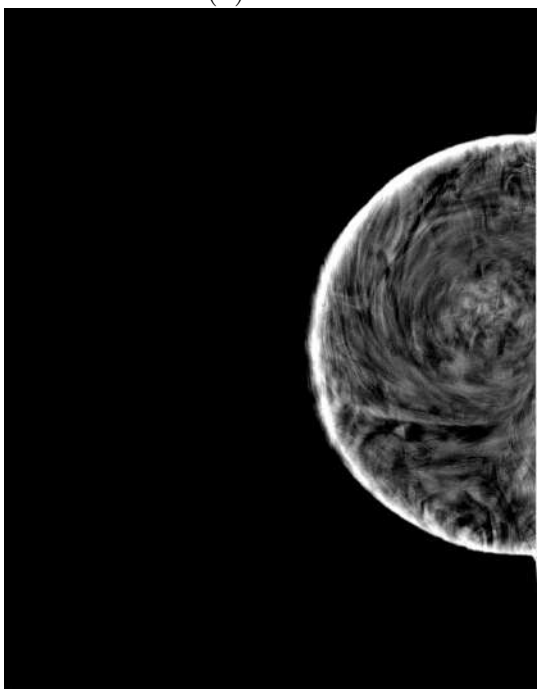


Tabela 6: Tempo (T_c), em segundos, que o algoritmo de controle demorou para ser executado.

Phantom	Média	Variância	Desvio Padrão
Shepp-Logan	1,39	0,01	0,09
BR3D	204,18	21,15	4,60

4.1 Utilizando os recursos do MATLAB

Antes de se utilizar CUDA foram feitos testes mais simples, utilizando recursos de programação paralela do próprio MATLAB. A tabela 7 mostra os resultados da aplicação das métricas comparativas em cada projeção obtida, no caso todas as técnicas utilizadas para a obtenção das projeções usando as ferramentas de programação em GPU próprias do MATLAB resultaram nos mesmos valores em cada uma das projeções. Todos os resultados foram extremamente elevados, o que significa que as projeções podem até ser consideradas idênticas às de controle.

Tabela 7: Resultados da aplicação de métricas comparativas em cada uma das 9 projeções adquiridas usando a GPU diretamente no MATLAB com as suas correspondentes de controle.

	Shepp-Logan		BR3D	
Projeção	PSNR	SSIM	PSNR	SSIM
1	164,21	1,00	148,29	1,00
2	164,05	1,00	148,55	1,00
3	164,90	1,00	148,53	1,00
4	164,36	1,00	148,40	1,00
5	164,06	1,00	148,36	1,00
6	164,85	1,00	148,44	1,00
7	164,28	1,00	148,61	1,00
8	164,37	1,00	148,67	1,00
9	164,18	1,00	148,43	1,00

A tabela 8 apresenta o tempo de execução (T_1) obtido com o algoritmo de controle ao alocar memória na GPU com o uso da função "gpuArray", já o tempo (T_2) da tabela 9 é relativo não só ao uso da função "gpuArray", mas também da função "arrayfun".

Tabela 8: Tempo (T_1), em segundos, que o algoritmo de controle demorou para ser executado ao usar *gpuArray* como tipo de variável de entrada.

Phantom	Média	Variância	Desvio Padrão	(T_c/T_1)
Shepp-Logan	0,71	0,00	0,02	1,96
BR3D	98,00	1,58	1,26	2,08

Tabela 9: Tempo (T_2), em segundos, que o algoritmo de controle demorou para ser executado ao usar a função "arrayfun"para calcular X e Y.

Phantom	Média	Variância	Desvio Padrão	(T_c/T_2)
Shepp-Logan	0,95	0,00	0,02	1,47
BR3D	15,59	0,00	0,03	13,10

Apesar da simplicidade desse processo, os resultados se mostraram bastante positivos, reduzindo praticamente a metade o tempo de execução no uso de ambos os *phantoms* somente ao disponibilizar para o MATLAB memória alocada na GPU. O uso da função "arrayfun"por outro lado teve resultados mais contrastantes entre os *phantoms*, acelerando muito mais o *phantom* BR3D que o de Shepp-logan.

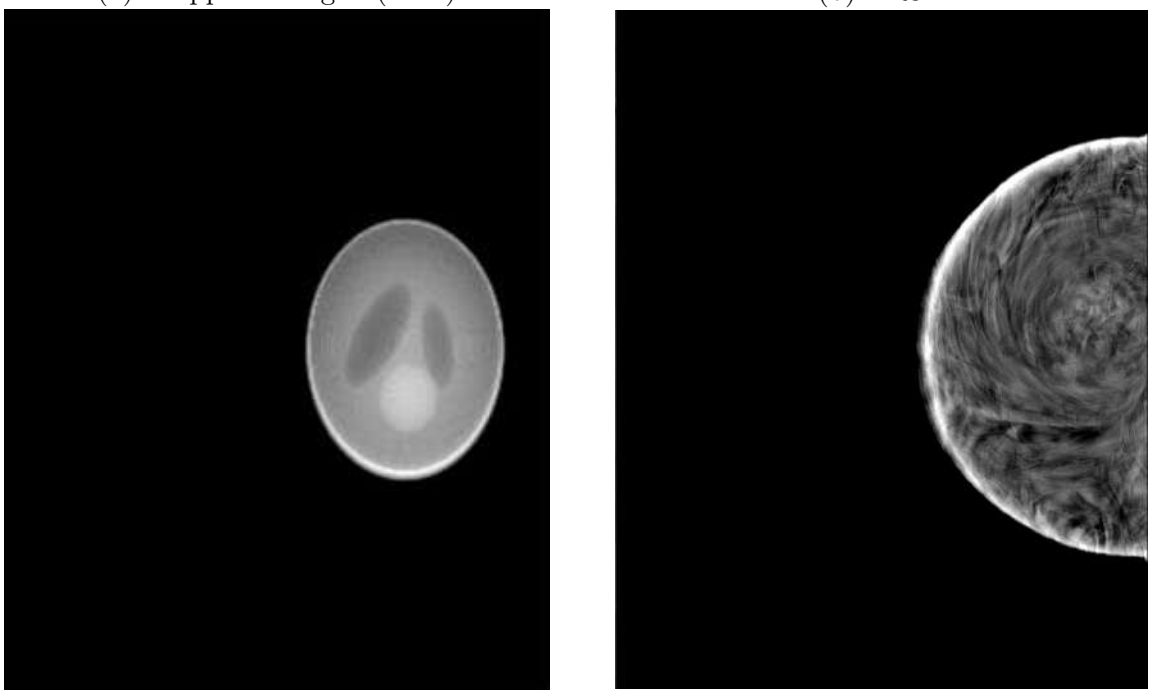
Isso se deve ao fato de que o *phantom* de Shepp-Logan é consideravelmente pequeno e consequentemente o tempo de execução do algoritmo também é, por isso o tempo acrescido pelo uso da função "arrayfun"foi proporcionalmente maior que o do *phantom* BR3D, sendo perceptível em um, mas não no outro.

4.2 Utilizando CUDA

Para utilizar CUDA foi necessário primeiro adaptar o código original escrito em MATLAB para C++, utilizando algumas das peculiaridades exigidas pelos arquivos *MEX* descridas na seção de Materiais e Métodos. Uma das projeções resultantes de cada *phantom* pode ser vista na figura 19.

Figura 19: Projeções obtidas com o algoritmo feito em C++.

(a) Shepp and Logan (1974) (b) BR3D

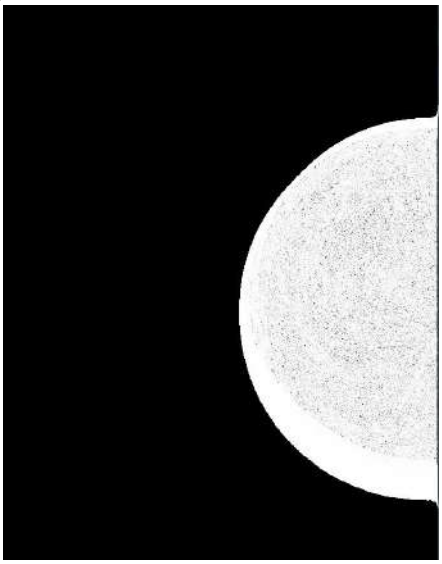


Embora as projeções sejam visualmente muito parecidas com as de controle, o mapa de SSIM deixa claro que não são realmente idênticas. Apesar disso as diferenças são tão sutis que, para melhor visualização, foi necessário ajustar seu contraste manualmente, conforme a figura 20. Para ilustrar os valores reais do mapa de SSIM, sem ajuste de contraste, foi plotado também um histograma também.

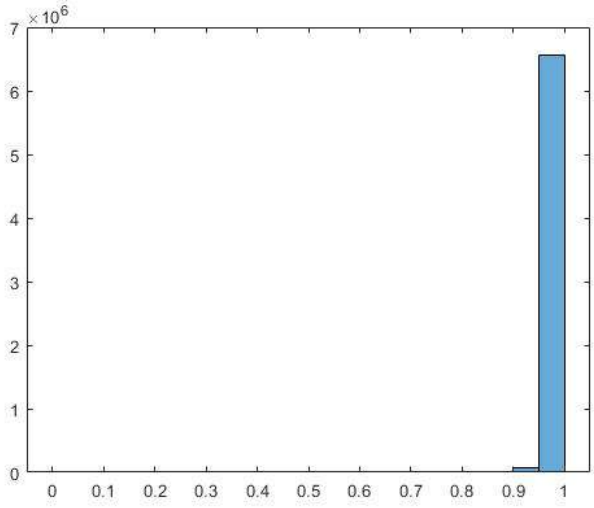
A fim de evitar que os reais valores da métrica SSIM fossem disfarçados também foi aplicada uma máscara com a forma do *phantom* sobre a projeção, uma vez que a área de interesse das projeções é uma região menor da imagem. A métrica PSNR, por sua vez, foi aplicada em recortes retangulares que enquadraram a área de interesse. Esses procedimentos foram tomados na aquisição das métricas de comparação de imagem para todas as projeções de ambos os *phantoms*.

Figura 20: Mapa de SSIM de uma das projeções do *phantom* BR3D, feita em C++.

(a) Mapa de SSIM com máscara e ajuste de contraste.



(b) Histograma sem ajuste de contraste.



A tabela 10 por sua vez mostra os resultados da aplicação das métricas comparativas em cada projeção obtida e, apesar de não serem tão elevados quanto antes, ainda demonstram um alto grau de proximidade.

Tabela 10: Resultados da aplicação de métricas comparativas em cada uma das 9 projeções adquiridas com os métodos que usaram C++, mas não usaram *shared memory*, com as suas correspondentes de controle.

	Shepp-Logan		BR3D	
Projeção	PSNR	SSIM	PSNR	SSIM
1	42,43	0,97	55,33	0,99
2	42,02	0,97	55,60	0,99
3	41,61	0,97	55,70	0,99
4	41,35	0,97	55,58	0,99
5	40,73	0,97	55,41	0,99
6	40,32	0,97	55,55	0,99
7	40,09	0,97	55,59	0,99
8	39,80	0,97	55,48	0,99
9	39,87	0,96	55,14	0,99

Analisando os resultados é possível inferir que se tratam de erros de arredondamento, uma vez que as projeções do *phantom* BR3D, cujo valor dos *pixels* possui mais casas antes da virgula, sofreu bem menos influência. Apesar de parte dessa diferença no arredondamento poder ser simples consequência da utilização de diferenças na linguagem MATLAB e C++, o maior responsável por isso é o uso de ponto flutuante de precisão simples no código em C++.

A escolha de se utilizar precisão simples foi devida a maior quantidades de núcleos de processamento desse tipo na GPU utilizada nos testes, bem como a baixa influência dessa conversão nas imagens, como mostrado anteriormente. A tabela 11 mostra os resultados obtidos com a execução do algoritmo escrito em C++. Nela é possível reparar que a simples conversão de linguagem já melhorou o desempenho do algoritmo e isso também se deve principalmente a utilização de ponto flutuante de precisão simples nos cálculos. Essa modificação foi feita pois a placa de vídeo utilizada possui mais núcleos para cálculos de ponto flutuante de precisão simples do que de precisão dupla.

Tabela 11: Tempo (T_3), em segundos, que o algoritmo feito em C++ demorou para ser executado.

Phantom	Média	Variância	Desvio Padrão	(T_c/T_3)
Shepp-Logan	0,76	0,00	0,00	1,83
BR3D	51,34	0,01	0,10	3,98

Os testes seguintes foram feitos adaptando este código de forma a permitir que fosse executado em GPU, mas mantendo a qualidade das projeções. As tabelas 12 a 14 demonstram os resultados obtidos adaptando os cálculos de aquisição das coordenadas X e Y e de interpolação para serem feitos de forma paralela com CUDA. Nesta etapa ainda foi utilizado um *loop* para as fatias, ou seja, os cálculos ainda foram feitos uma fatia de cada vez.

Tabela 12: Tempo (T_4), em segundos, que o algoritmo feito em C++ demorou para ser executado usando CUDA somente nos cálculos da interpolação, fatia por fatia.

Phantom	Média	Variância	Desvio Padrão	(T_c/T_4)
Shepp-Logan	3,70	0,00	0,05	0,38
BR3D	268,08	2,09	1,44	0,76

Tabela 13: Tempo (T_5), em segundos, que o algoritmo feito em C++ demorou para ser executado usando CUDA somente nos cálculos de X e Y, fatia por fatia.

Phantom	Média	Variância	Desvio Padrão	(T_c/T_5)
Shepp-Logan	1,08	0,00	0,00	1,29
BR3D	50,42	0,00	0,06	4,05

Tabela 14: Tempo (T_6), em segundos, que o algoritmo feito em C++ demorou para ser executado usando CUDA nos cálculos de X, Y e interpolação, fatia por fatia.

Phantom	Média	Variância	Desvio Padrão	(T_c/T_6)
Shepp-Logan	0,62	0,00	0,01	2,23
BR3D	34,57	0,01	0,08	5,90

Pode-se observar com esses dados que, quando a GPU não é utilizada para realizar quantidades realmente muito grandes de cálculos simultâneos, ela pode não apresentar ganhos significativos em relação a execução na CPU, podendo inclusive demorar ainda mais para fazê-los. Em todos os três casos o desempenho da execução com *phantom* BR3D foi melhor em relação ao *phantom* de Shepp-Logan justamente por proporcionar maior quantidade de cálculos a cada etapa.

Também é possível perceber que a execução dos cálculos de coordenadas de forma paralela melhorou mais o desempenho geral do algoritmo que o cálculo da interpolação, mesmo o número de *threads* simultâneas exigidas para tal sendo menor, o que significa que a complexidade dos cálculos efetuados em paralelo também faz bastante diferença no desempenho.

O algoritmo que efetuou ambos os cálculos em paralelo obteve os melhores resultados, por esse motivo também foi aplicado na aquisição dos dados da tabela 16, com a diferença que o último também utilizou de *shared memory*. Apesar da variação no algoritmo C++ em si não ter visualmente resultado em mudança significativa nas projeções, a utilização das métricas apontou uma ligeira mudança, conforme demonstra a tabela 15.

Tabela 15: Resultados da aplicação de métricas comparativas em cada uma das 9 projeções adquiridas com os métodos que usaram C++ e *shared memory* com as suas correspondentes de controle.

	Shepp-Logan		BR3D	
Projeção	PSNR	SSIM	PSNR	SSIM
1	42,80	0,96	55,35	0,99
2	42,38	0,96	55,62	0,99
3	42,03	0,96	55,71	0,99
4	41,74	0,97	55,60	0,99
5	41,14	0,96	55,43	0,99
6	40,73	0,97	55,58	0,99
7	40,51	0,96	55,62	0,99
8	40,21	0,96	55,52	0,99
9	40,24	0,96	55,18	0,99

Tabela 16: Tempo (T_7), em segundos, que o algoritmo feito em C++ demorou para ser executado usando CUDA nos cálculos de X, Y e interpolação, fatia por fatia, com uso de *shared memory*.

Phantom	Média	Variância	Desvio Padrão	(T_c/T_7)	(T_6/T_7)
Shepp-Logan	0,36	0,00	0,00	3,83	1,72
BR3D	19,15	0,01	0,10	10,66	1,80

O simples uso de *shared memory* foi suficiente para melhorar a velocidade de execução do mesmo algoritmo em torno de 1.75 vezes, mesmo quando a quantidade de acessos simultâneos a endereços de memória não era tão elevada (caso do *phantom* de Shepp-Logan). Isso aconteceu não só porque a *shared memory* fica fisicamente mais próxima dos SMs, mas também porque ela por si só é composta de registradores mais rápidos que a DRAM da *global memory* e seu uso alivia o número de acessos da DRAM.

A foma como a *shared memory* armazena dados também foi suficiente para alterar ligeiramente as imagens resultantes e, embora tal diferença seja diminuta, é interessante perceber como a memória utilizada influencia nos cálculos.

As tabelas 17 e 18 por sua vez apresentam os resultados dos testes feitos com esse mesmo algoritmo ao utilizar *texture memory* em vez e *shared memory*. Não foi possível obter projeções satisfatórias com *texture memory* utilizando o *phantom* BR3D porque a quantidade desse tipo de memória disponível na placa gráfica empregada não foi suficiente para armazenar uma única fatia do *phantom*, por isso as conclusões tomadas serão referentes apenas ao uso do *phantom* de Shepp-Logan.

Tabela 17: Tempo (T_8), em segundos, que o algoritmo feito em C++ demorou para ser executado usando CUDA nos cálculos de X, Y e interpolação, fatia por fatia, com uso de *texture memory*.

Phantom	Média	Variância	Desvio Padrão	(T_c/T_8)	(T_6/T_8)
Shepp-Logan	0,59	0,00	0,01	2,36	1,05
BR3D	-	-	-	-	-

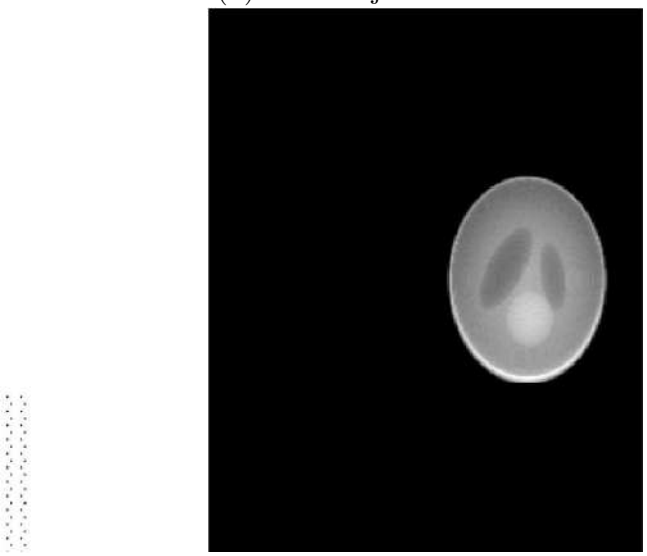
Tabela 18: Tempo (T_9), em segundos, que o algoritmo feito em C++ demorou para ser executado usando CUDA nos cálculos de X, Y e interpolação, fatia por fatia, com uso de *texture memory* diretamente no cálculo da interpolação.

Phantom	Média	Variância	Desvio Padrão	(T_c/T_9)	(T_6/T_9)	(T_8/T_9)
Shepp-Logan	0,54	0,00	0,01	2,55	1,14	1,08
BR3D	-	-	-	-	-	-

Nenhum dos métodos acelerou a execução em mais do que 1,15 vezes em relação ao T_6, obtido empregando o mesmo algoritmo, mas sem fazer uso de *texture memory*. Ao empregar a *texture memory* apenas para acessos de memória, mantendo o uso do algoritmo 2 nos cálculos de interpolação, as métricas de qualidade de imagem apontaram os mesmos resultados dos algoritmos feitos em C++ apresentados anteriormente. Porém ao usar a ferramenta de interpolação própria da *texture memory* parte das projeções foi corrompida, contendo valores esdrúxulos, isso quando tais valores ainda eram lidos como valor numérico pelo MATLAB e não NaN (*not a number*).

Figura 21: Exemplo de projeção obtida com uso da ferramenta de interpolação própria da *texture memory*.

(a) Escala ajustada automaticamente. (b) Escala ajustada entre 0 e 35.

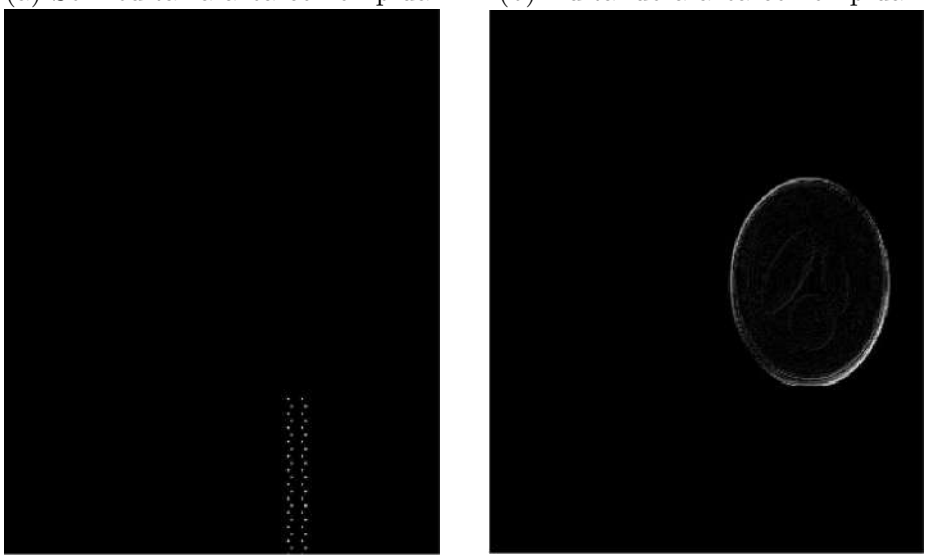


A figura 21 mostra lado a lado uma mesma projeção obtida usando a ferramenta de cálculo de interpolação da *texture memory* plotadas de duas formas distintas, uma deixando o intervalo de intensidade entre os *pixels* pretos e brancos ser ajustado automaticamente para compreender os valores máximo e mínimo presentes na imagem, enquanto a outra foi ajustada para que todos os *pixels* cujo valor é menor que zero seja completamente preto.

A figura 22 por sua vez ilustra a diferença entre a projeção mostrada na figura 21 com sua correspondente de controle, com e sem editar os *pixels* da área corrompida.

Figura 22: Diferença entre a projeção da figura 21 e sua correspondente de controle.

(a) Sem editar a área corrompida. (b) Editando a área corrompida.



Ao tentar aplicar as métricas PSNR e SSIM para avaliar a qualidade de projeções como as apresentadas acima, as funções próprias do MATLAB, que também foram empregadas nos demais testes, simplesmente falharam em chegar a algum valor conclusivo. Apesar disso, a própria NVIDIA (2020) na documentação do *CUDA toolkit* enfatiza que a interpolação feita pela *texture memory* é de baixa precisão, então mesmo na ausência desse problema os resultados provavelmente teriam métricas inferiores.

Sabendo que não houveram problemas ao empregar a *texture memory* apenas para acessos de memória, o que resultou inclusive em imagens de qualidade equivalente a dos demais métodos testados, é possível inferir que o problema não ocorreu por falta de memória.

Posteriormente o algoritmo foi modificado para se aumentar ainda mais seu grau de paralelismo, de forma que todos os cálculos das fatias foram feitos simultaneamente a cada execução do *loop* das projeções. Os resultados obtidos nos testes desse algoritmo, usando e sem usar *shared memory*, se encontram nas tabelas 19 e 20.

Tabela 19: Tempo (T_{10}), em segundos, que o algoritmo feito em C++ demorou para ser executado usando CUDA nos cálculos de X, Y e interpolação, todas as fatias simultaneamente, sem usar *shared memory*.

Phantom	Média	Variância	Desvio Padrão	(T_c/T_{10})	(T_6/T_{10})
Shepp-Logan	0,05	0,00	0,00	29,33	13,12
BR3D	1,99	0,00	0,01	102,67	17,39

Tabela 20: Tempo (T_{11}), em segundos, que o algoritmo feito em C++ demorou para ser executado usando CUDA nos cálculos de X, Y e interpolação, todas as fatias simultaneamente, usando *shared memory*.

Phantom	Média	Variância	Desvio Padrão	$(T_c/T11)$	(T_{10}/T_{11})
Shepp-Logan	0,05	0,00	0,00	26,28	0,90
BR3D	2,67	0,00	0,01	76,40	0,74

Aumentar o grau de paralelismo do algoritmo reduziu drasticamente seu tempo de execução, chegando a aumentar sua velocidade em até 17 vezes em relação a sua versão anterior e 102 vezes em relação ao algoritmo de controle. Desta vez, porém, o uso de *shared memory* prejudicou seu desempenho.

Isso aconteceu porque, ao aumentar o grau de paralelismo do algoritmo, o tempo levado para cada acesso de memória fez menos diferença, uma vez que agora mais acessos são efetuados simultaneamente. Além disso, para usar *shared memory* foi necessário aplicar primitivas de sincronização (barreiras) na execução das *threads*, o que provoca atrasos que diminuem a velocidade do algoritmo. Contudo, se os algoritmos em cada *thread* fossem complexos ao ponto de exigir uma quantidade bem maior de acessos de memória, o resultado seria diferente.

Tabela 21: Tempo (T_{12}), em segundos, que o algoritmo feito em C++ demorou para ser executado usando CUDA nos cálculos de X, Y e interpolação, todas as fatias simultaneamente, alterando a disposição das *threads* e *thread blocks*.

Phantom	Média	Variância	Desvio Padrão	(T_c/T_{12})	(T_{10}/T_{12})
Shepp-Logan	0,02	0,00	0,00	72,18	2,46
BR3D	1,10	0,00	0,01	186,29	1,81

Os resultados que se encontram na tabela 21, por sua vez, são referentes ao mesmo algoritmo utilizado na tabela 19, efetuando os cálculos de todas as fatias de uma vez e sem usar *shared memory*. A diferença aqui é a forma como as *threads* e *thread blocks* foram organizados na *grid*. Nas versões anteriores tal organização variava bastante em cada chamada de cada *kernel*, a fim de facilitar o entendimento do código e seu desenvolvimento, já o código da tabela 21 mantém uma melhor consistência nessa organização, lançando sempre uma quantidade de *thread blocks* na dimensão x maior do que nas dimensões y e z.

Essa mudança foi feita inicialmente para garantir que a placa de vídeo utilizada conseguisse instanciar a quantidade de *thread block* necessárias ao executar todas as projeções simultaneamente, isso porque a dimensão x é aquela que permite maior quantidade de *thread blocks*, conforme demonstra a tabela 4. Entretanto pôde-se perceber com essa mudança que, até mesmo a simples disposição das *threads* na *grid* pode fazer uma diferença considerável no resultado final.

Tabela 22: Tempo (T_{13}), em segundos, que o algoritmo feito em C++ demorou para ser executado usando CUDA nos cálculos de X, Y e interpolação, todas as projeções simultaneamente.

Phantom	Média	Variância	Desvio Padrão	(T_c/T_{13})	(T_{12}/T_{13})
Shepp-Logan	0,02	0,00	0,00	69,31	0,96
BR3D	1,10	0,00	0,01	185,03	0,96

O algoritmo utilizado para obter os resultado da tabela 22 foi feito sem a presença de *loop* algum, a fim de executar os cálculos de todas as fatias de todas as projeções ao mesmo tempo. Neste caso, entretanto, aumentar o grau de paralelismo do código desta vez não melhorou seu desempenho, na verdade o piorou ligeiramente.

Ao executar os cálculos de todas as projeções simultaneamente, muitas *threads* requisitaram acesso aos mesmos endereços de memória, congestionando os barramentos da DRAM e fazendo com que muitas *threads* tivessem que esperar as outras terminarem suas tarefas antes que pudessem, sequer efetuar a leitura de memória.

Outro fator que pode ter contribuído para esse resultado é o uso da *atomic function* que efetuou os somatórios. Apesar desta mesma função ter sido utilizada em todos os outros algoritmos, a maior quantidade de acessos que esta função teve que gerenciar possivelmente agravou as consequências de sua natureza sincronizada.

A tabela 23 apresenta os resultados obtidos ao empregar um algoritmo similar ao 3, mas com a diferença de que executa todas as projeções simultaneamente e uma fatia de cada vez. Como existem mais fatias que projeções, apenas efetuar essa mudança reduziria significativamente o grau de paralelismo do algoritmo, então os cálculos de todas a coordenadas X e Y foram feitos de forma paralela e fora do *loop* das fatias.

É interessante ver como essa abordagem resultou em um tempo de execução menor que o T_{10} quando foi usado o *phantom* menor, mas ao usar o *phantom* BR3D os tempos foram praticamente iguais.

Tabela 23: Tempo (T_{14}), em segundos, que o algoritmo feito em C++ demorou para ser executado usando CUDA nos cálculos de X, Y e interpolação, todas as projeções simultaneamente, mas uma fatia de cada vez.

Phantom	Média	Variância	Desvio Padrão	(T_c/T_{14})	(T_{10}/T_{14})
Shepp-Logan	0,04	0,00	0,00	37,96	1,29
BR3D	1,98	0,00	0,01	102,91	1,00

A tabela 24 mostra os resultados executando o algoritmo da tabela 23, mas desta vez sobrepondo execução e transferência e dados ao usar múltiplas *streams*, o que não foi suficiente para melhorar os resultados, em vez disso os piorou ligeiramente, consequência do custo computacional agregado para se criar e gerenciar as *streams*. Além disso, o tempo consumido na execução dos cálculos pode ter sido consideravelmente maior que o tempo da transferência de dados, tornando a econimia de tempo mínima.

Tabela 24: Tempo (T_{15}), em segundos, que o algoritmo feito em C++ demorou para ser executado usando CUDA nos cálculos de X, Y e interpolação, todas as projeções simultaneamente, mas uma fatia de cada vez, utilizando múltiplas *streams*.

Phantom	Média	Variância	Desvio Padrão	(T_c/T_{15})	(T_{14}/T_{15})
Shepp-Logan	0,04	0,00	0,00	33,89	0,89
BR3D	2,17	0,00	0,02	93,91	0,91

Por fim a tabela 25 apresenta um resumo dos resultados obtidos ao empregar cada um dos métodos testados neste trabalho, enquanto as figuras 23 e 24, referentes ao *phantom* de Shepp-Logan e ao *phantom* BR3D respectivamente, ilustram tais resultados de forma gráfica.

Tabela 25: Tempo de execução obtido em cada método.

Nome	**Descrição**	**Shepp-Logan**	**BR3D**
Tc	Original da toolbox	1.39s	204.18s
T1	Usando gpuArray no MATLAB	0.71s	98.00s
T2	Usando arrayfun no MATLAB	0.95s	15.59s
T3	C++ sem CUDA	0.76s	51.34s
T4	Cálculo da interpolação com CUDA	3.70s	268.08s
T5	Cálculo das coordenadas com CUDA	1.08s	50.42s
T6	Cálculos da interpolação e das coordenadas com CUDA	0.62s	34.57s
T7	Algoritmo T6 usando shared memory	0.36s	19.15s
T8	Algoritmo T6 usando texture memory apenas para acessar dados	0.59s	-
T9	Algoritmo T6 usando texture memory no cálculo da interpolação	0.54s	-
T10	Efetuando todos os cálculos de todas as fatias em paralelo com CUDA	0.05s	1.99s
T11	Algoritmo T10 usando shared memory	0.05s	2.67s
T12	Algoritmo T10 alterando a organização das threads	0.02s	1.10s
T13	Efetuando todas as projeções em paralelo com CUDA	0.02s	1.10s
T14	Algoritmo 3 efetuando todas as projeções em paralelo com CUDA	0.04s	1.98s
T15	Algoritmo T14 usando 32 streams	0.04s	2.17s

Figura 23: Tempos de execução do *phantom* de Shepp-Logan para cada método testado.

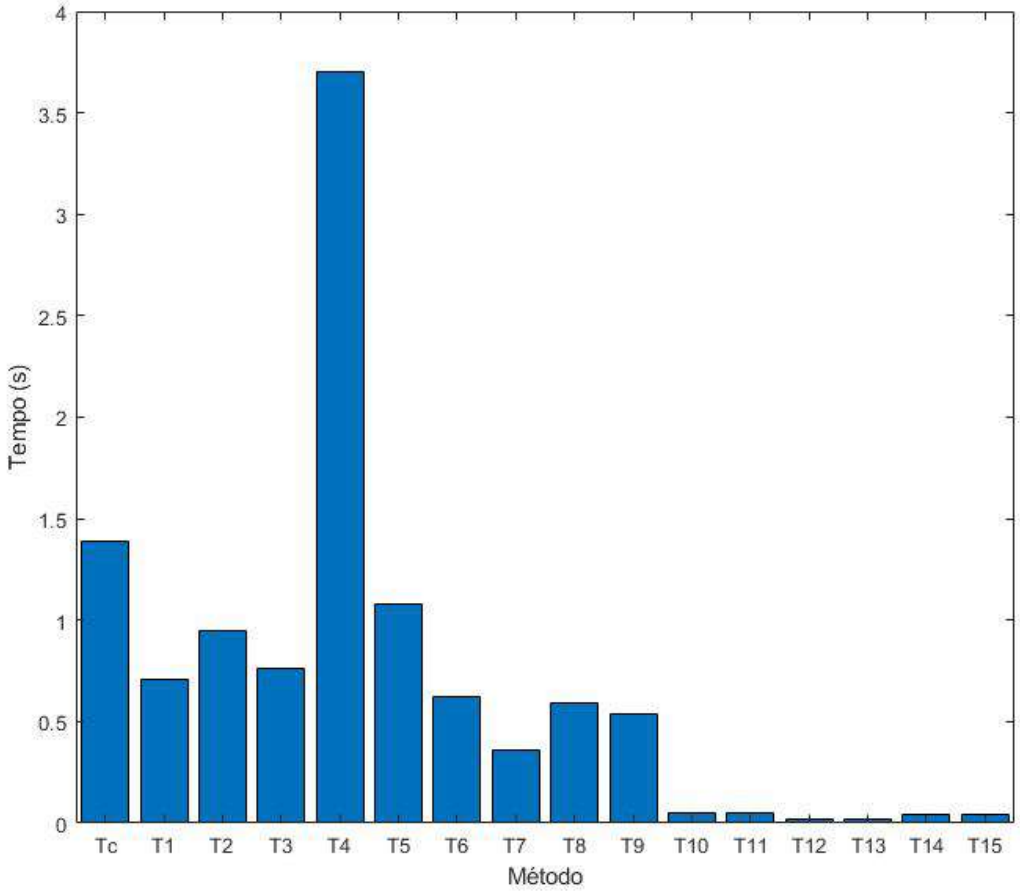
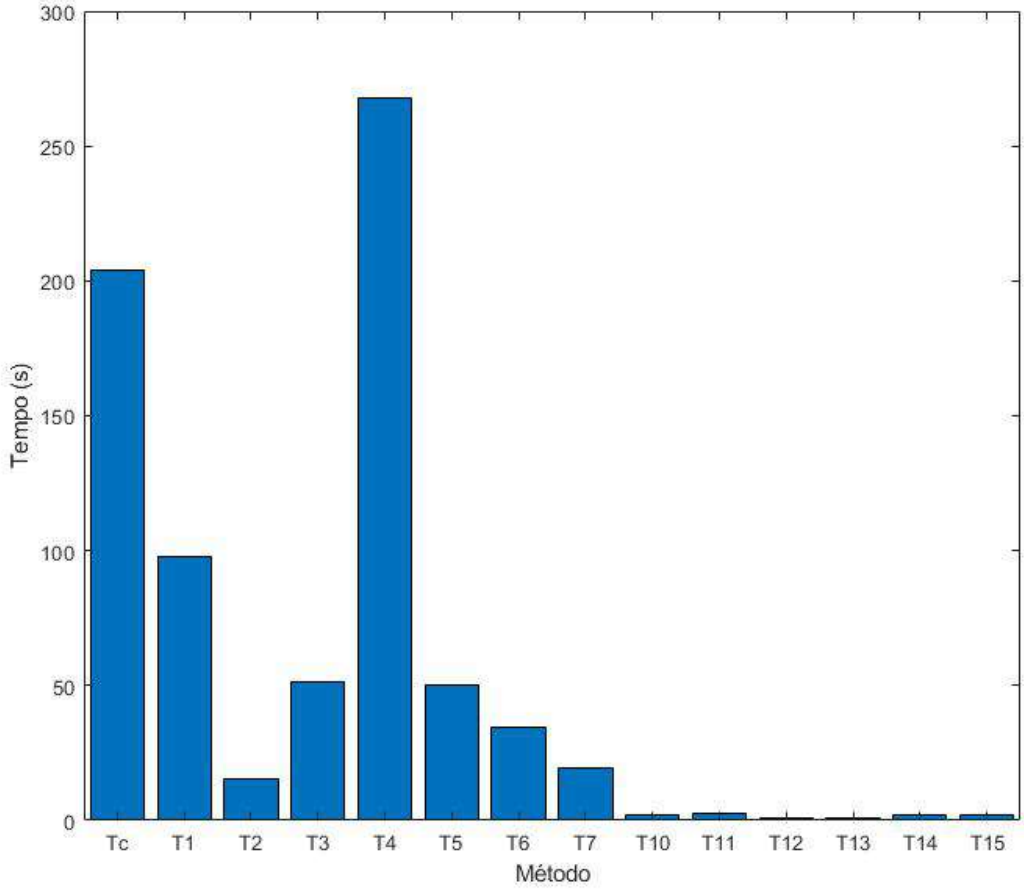


Figura 24: Tempos de execução do *phantom* BR3D para cada método testado.



5 CONCLUSÃO

O presente trabalho teve como principal objetivo investigar o potencial computacional das placas gráficas, utilizando como objeto de estudo uma *toolbox* desenvolvida para reconstrução de imagens de tomossíntese digital mamária. A principal ferramenta empregada nesse estudo foi GPU, utilizada majoritariamente com CUDA, mas como a *toolbox* foi desenvolvida em MATLAB, algumas das funcionalidades deste *software* também foram testadas.

Apesar da simplicidade das ferramentas de programação em GPU do MATLAB, elas se mostraram muito mais eficientes que o esperado, chegando a acelerar o algoritmo de projeção em até 13 vezes. Embora esse não seja todo o potencial das GPU, como mostrado no decorrer deste trabalho, esta metodologia ainda proporcionou um ganho de tempo considerável sem a necessidade de empregar grande esforço para isso.

O uso de CUDA, por sua vez, exigiu muito mais atenção e empenho, as vezes aumentando o tempo de execução do algoritmo em vez de diminuí-lo, mas quando usado de maneira certa foi capaz de acelerar sua execução em até 186 vezes. No decorrer dos testes ficou claro que, de modo geral, alterações na forma como os algoritmos são escritos influenciam muito mais no seu tempo de execução do que o uso de ferramentas como *streams* e *texture objects*.

Isso acontece porque, diferentemente das CPUs, as GPUs são projetadas para executar grandes quantidades de operações independentes entre si a cada ciclo de trabalho, então é esperado que modificar a estrutura dos algoritmos a fim de aumentar seu grau de paralelismo acelere a execução. Contudo existem muitos outros fatores que interferem na eficiência de um código CUDA, o que torna difícil prever exatamente que tipo de abordagem é a mais adequada sem testar as opções.

Alguns dos fatores que devem ser levados em consideração na hora de escrever um código em CUDA são a complexidade dos algoritmos de cada *kernel*, a organização das *threads* na *grid* e se diferentes *threads* requisitam acesso a um mesmo endereço de memória. Contudo o uso de ferramentas mais específicas também pode ter um impacto considerável no tempo de execução do algoritmo quando aplicadas nas circunstancias certas, como demonstrado nos testes feitos com *shared memory*.

REFERÊNCIAS

Adve, S. V., Adve, V. S., et al. (2008). Parallel computing research at illinois. University of Illinois at Urbana-Champaign.

Ahmed, K. and Schuegraf, K. (2011). Transistor Wars Rival: architectures face off in a bid to keep Moore's Law alive. *IEEE Spectrum*, pages 50–66.

Arora, M. (2012). The Architecture and Evolution of CPU-GPU Systems for General Purpose Computing. *Department of Computer Science and Engineering University of California, San Diego*, 27.

ASC (2020). American Cancer Society. `https://www.cancer.org/` Acesso em: 24 de abr. de 2020.

Brodtkorb, A. R., Hagen, T. R., and Sætra, M. L. (2013). Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, 73:4–13.

CIRS (2013). BR3D Breast Imaging Phantom Model 020. `www.cirsinc.com/products/all/51/br3d-breast-imaging-phantom` Acesso em: 24 de nov. de 2018.

Denning, P. J. and Lewis, T. G. (2017). Exponential laws of computing growth. *Communications of the ACM*, 60:54–65.

Geer, D. (2005). Chip makers turn to multicore processors. *IEEE Computer*, 38:11–13.

Huang, W., Stan, M. R., et al. (2010). Interaction of scaling trends in processor architecture and cooling. In *26th Annual IEEE Semiconductor Thermal Measurement and Management Symposium (SEMI-THERM)*. IEEE.

INCA (2020). Instituto Nacional de Câncer José Alencar Gomes da Silva. `https://www.inca.gov.br/` Acesso em: 23 de abr. de 2020.

Keckler, S. W., Olukotun, K., and Hofstee, H. P. (2009). *Integrated Circuits and Systems*. Springer.

Kim, H., Vuduc, R., Baghsorkhi, S., Choi, J., and mei Hwu, W. (2012). Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU). *Morgan and Claypool Publishers*.

K.T. Gribbon, C.T. Johnston, and D.G. Bailey (2003). A Real-time FPGA Implementation of a Barrel Distortion Correction Algorithm with Bilinear Interpolation .

Loh, G. H., Xie, Y., and Black, B. (2007). Processor design in 3D die-stacking technologies. *IEEE Micro*, pages 31–48.

MathWorks (2020). Matlab documentation. `https://www.mathworks.com/help/matlab/index.html?s_tid=CRUX_lftnav` Acesso em: 10 de mai. de 2020.

McClanahan, C. (2010). History and Evolution of GPU Architecture.

McCool, M., Robison, A. D., and Reinders, J. (2012). *Structured Parallel Programming*. Elsevier.

Nickolls, J. and Dally, W. J. (2010). The GPU Computing Era. *IEEE Computer Society*.

NVIDIA (2016). Nvidia tesla p100. `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf` Acesso em: 5 de mai. de 2020.

NVIDIA (2020). CUDA Toolkit Documentation. `https://docs.nvidia.com/cuda/` Acesso em: 10 de mai. de 2020.

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. (2008). GPU Computing. *Proceedings of the IEEE*.

Ramanathan, R. M., Thomas, V., et al. (2015). Platform 2015:intel processor and platform evolution for the next decade. Technical report, Intel.

SCHABEL, M. (2006). 3D Shepp-Logan phantom. Mathworks - File Exchange. `www.mathworks.com/matlabcentral/fileexchange/9416-3d-shepp-logan-phantom` Acesso em: 26 de nov. de 2018.

Shah, U. A. and Yousaf, S. (2019). Performance analysis of benchmarks for gpu-based linear programming problem solvers. In *2nd International Conference on Communication, Computing and Digital systems (C-CODE)*. IEEE.

Shepp, L. A. and Logan, B. F. (1974). The fourier reconstruction of a head section. *IEEE Transactions on Nuclear Science*, pages 21–43.

Theis, T. N. and Solomon, P. M. (2010). It's time to reinvent the transistor! *Science*, 327.

Vedantham S. and et al (2015). *Digital breast tomosynthesis: state of the art*. Radiology, Radiological Society of North America.

Vimieiro, R. B., Borges, L. R., Caron, R. F., Barufaldi, B., Bakic, P. R., Maidment, A. D. A., and Vieira, M. A. C. (2019). Noise measurements from reconstructed digital breast tomosynthesis. In *Medical Imaging 2019: Physics of Medical Imaging*, volume 10948.

Vimieiro R. B., Borges L. R., and Vieira, M. A. C. (2019). Open source reconstruction toolbox for digital breast tomosynthesis. In: Costa-Felix R., Machado J., Alvarenga A. (eds) XXVI Brazilian Congress on Biomedical Engeneering. IFMBE Proceedings, vol 70/2. Springer, Singapore.

Vimieiro, Rodrigo de Barros (2019). Ferramenta para reconstrução de imagens de tomossíntese mamária e sua aplicação na análise do ruído em imagens reconstruídas. Dissertação (Mestrado em Processamento de Sinais de Instrumentação) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2019. doi: 10.11606/D.18.2019.tde-04042019-104114. Acesso em 2020-06-09.

Yao, K. (2004). *High-Frequency and High-Performance VRM Design for the Next Generations of Processors*. PhD thesis, Faculty of the Virginia Polytechnic Institute and State University.

Zhou Wang, Bovik, A. C., Sheikh, H. R., and Simoncelli, E. P. (2004). Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612.