



UNIVERSIDADE FEDERAL DO PARÁ  
INSTITUTO DE CIÊNCIAS EXATAS E NATURAIS  
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

NIELSEN ALVES GONÇALVES

**AVALIAÇÃO DE DESEMPENHO DO USO DE ACELERADORES GRÁFICOS NA  
SOLUÇÃO DE MÉTODOS E TÉCNICAS NUMÉRICAS**

Belém  
2016

Nielsen Alves Gonçalves

**AVALIAÇÃO DE DESEMPENHO DO USO DE ACELERADORES GRÁFICOS NA  
SOLUÇÃO DE MÉTODOS E TÉCNICAS NUMÉRICAS**

Dissertação de Mestrado apresentada ao  
Programa de Pós-Graduação em Ciência da  
Computação da Universidade Federal do Pará  
como requisito parcial para obtenção do título  
de Mestre em Ciência da Computação

Orientador:  
Prof. Dr. Josivaldo de Souza Araújo

Belém  
2016

**Dados Internacionais de Catalogação-na-Publicação (CIP)**  
**Biblioteca Central - UFPA**

---

Gonçalves, Nielsen Alves, 1984-

Avaliação de desempenho do uso de aceleradores gráficos na solução de métodos e técnicas numéricas / Nielsen Alves Gonçalves. — 2016

Orientador prof. Dr. Josivaldo de Souza Araújo

Dissertação (Mestrado) - Universidade Federal do Pará, Instituto de Ciências Exatas e Naturais, Programa de Pós-Graduação em Ciência da Computação, 2016.

1. Computação paralela (Computação). 2. CUDA (Arquitetura de computador). 3. Software de aplicação - Desenvolvimento. I. Título.

CDD - 23. ed. 005.275

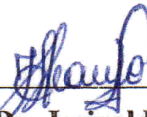
---

UNIVERSIDADE FEDERAL DO PARÁ  
INSTITUTO DE CIÊNCIAS EXATAS E NATURAIS  
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

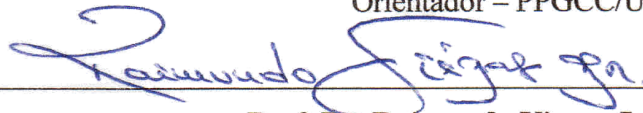
**NIELSEN ALVES GONÇALVES**

**AVALIAÇÃO DE DESEMPENHO DO USO DE ACELERADORES GRÁFICOS  
NA SOLUÇÃO DE MÉTODOS E TÉCNICAS NUMÉRICAS**

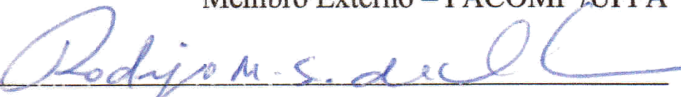
Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal do Pará como requisito para obtenção do título de Mestre em Ciência da Computação, defendida e aprovada em 22/12/2016, pela banca examinadora constituída pelos seguintes membros:



**Prof. Dr. Josivaldo de Souza Araújo**  
Orientador – PPGCC/UFPA

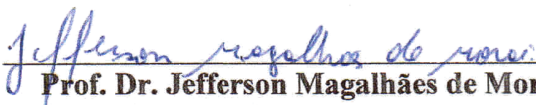


**Prof. Dr. Raimundo Viegas Junior**  
Membro Externo – FACOMP /UFPA



**Prof. Dr. Rodrigo Melo e Silva de Oliveira**  
Membro Externo – ITEC /UFPA

Visto:



**Prof. Dr. Jefferson Magalhães de Moraes**  
Coordenador do PPGCC/UFPA

*Prof. Dr. Jefferson Magalhães de Moraes*

Aos meus pais.

## AGRADECIMENTOS

Clichês só o são porque funcionam. Aqui vão alguns. Agradeço:

A Deus, pelo dom da vida.

Aos meus pais, Pedro e Lúcia, pela dedicação e amor incondicional que me permitiram chegar a mais esta etapa. Aos meus irmãos Diogo e Kelly, pelo carinho demonstrado das mais diversas formas.

À minha esposa, Patrícia, minha companheira, amiga, inspiradora, metade... que sempre acreditou, algumas vezes mais que eu, que eu seria capaz de concluir este trabalho.

Ao meu primo Fernando e família, pelo apoio, acolhimento e pelas longas e inspiradoras conversas sobre o fazer científico.

À Sandra, ao Professor Viégas, ao Antônio, ao Tulyo, ao Rafael, ao Lucas, à Luana e todos os demais colegas do Projeto SET, que compartilharam comigo muitos momentos de amizade e aprendizados que são parte importante da minha formação como pesquisador e como ser humano.

Ao professor Cícero, pela amizade, apoio e exemplo. Aos Professores Jessé e Ellen e aos demais colegas, alunos e professores da Geofísica da UFPA, de onde saíram as primeiras ideias para este trabalho, dentre tantos outros aprendizados.

Ao pessoal do LEMAG: Professor Rodrigo, Ramon, Washington e demais colegas, pela generosidade com que me acolheram durante o período em que frequentei o laboratório.

Ao colega de classe Daniel Souza pelas valiosas contribuições.

Ao Professor Josivaldo, por persistir na orientação deste trabalho mesmo após tantas dificuldades. Aos demais professores do PPGCC pelos ensinamentos e inspiração.

Aos demais amigos que cometi a injustiça de não citar nominalmente, mas que de várias outras formas contribuíram para a execução e conclusão deste trabalho e de mais esta fase da minha vida.

Obrigado!

*“Que homem é sábio? O que estuda sem cessar. Que homem é forte? O que sabe se dar limites. Que homem é rico? O que é feliz com o que tem.”.*  
(O Talmude)

## RESUMO

O desenvolvimento de plataformas computacionais heterogêneas tem tornado mais acessíveis os recursos capazes de aumentar o desempenho de aplicações, porém, tais recursos por vezes não oferecem a escalabilidade desejada, tornando necessário o uso de soluções distribuídas. No entanto, a implementação de soluções de computação heterogênea distribuída torna necessária a combinação de ferramentas de programação específicas, o que aumenta a complexidade e dificulta a implementação.

O presente trabalho apresenta avaliações de desempenho de implementações de métodos comumente utilizados na Computação Científica: O método da Eliminação Gaussiana, implementado utilizando um modelo de programação para Aceleradores baseado em diretivas; e os Métodos de Diferenças Finitas e Diferenças Finitas no Domínio do Tempo, implementados utilizando múltiplos aceleradores por meio de uma arquitetura de rede cliente/servidor, que simplifica o processo de programação de uma solução heterogênea distribuída ao permitir a abstração da camada de rede ao programador. Os experimentos descritos neste trabalho foram executados em hardware *commodity*, com o objetivo de avaliar a performance da referida arquitetura nesta categoria de hardware.

Os resultados obtidos com os experimentos indicam a possibilidade de ampliação de ganhos de desempenho no desenvolvimento de aplicações científicas, mesmo em hardware de baixo custo, por meio do uso de novos recursos de computação heterogênea em substituição a ferramentas baseadas em CPU.

**Palavras-chave:** CUDA. OpenACC. rCUDA. FDM.



## ABSTRACT

The heterogeneous computing platforms development have been made more accessible the resources that can increase the performance of applications, however, such resources often do not provide the desired scalability, motivating the use of distributed solutions. However, the implementation of heterogeneous distributed computing solutions makes necessary the combination of specific programming tools, which increases the complexity and complicates the implementation.

This work presents evaluations of performance of method implementations commonly used in Scientific Computing: The method of Gaussian elimination, implemented using a Directive based programming model for Accelerators; and the Finite Differences Time Domain Methods, both implemented using multiple accelerators through a client/server network architecture, which simplifies the process of programming a heterogeneous distributed solution by allowing the abstraction of the network layer to the programmer. The experiments described in this paper were run on commodity hardware, in order to evaluate the performance of such architecture in that hardware category.

The results obtained from experiments indicates the possibility of performance improvements in scientific applications development, even in low cost hardware, through the use of new heterogeneous computational resources instead of CPU based tools.

**Keywords:** CUDA. OpenACC. rCUDA. FDM.

## LISTA DE FIGURAS

Figura 1 – Padrão <i>Fork-Join</i> utilizado no OpenMP (Fonte: Elaborada pelo autor) . . .	18
Figura 2 – Representação de um sistema em Memória Distribuída (adaptado de (COOK, 2013)) . . . . .	19
Figura 3 – Arquitetura básica de um sistema heterogêneo CPU/GPU (Fonte: Elaborada pelo autor) . . . . .	20
Figura 4 – Arquitetura CUDA (Fonte: (NVIDIA, 2013)) . . . . .	21
Figura 5 – Representação de uma GPGPU (adaptado de (COOK, 2013)) . . . . .	22
Figura 6 – Diagrama da arquitetura rCUDA (Adaptado de (RCUDA, 2016)) . . . . .	25
Figura 7 – Representação de um ambiente rCUDA . . . . .	26
Figura 8 – Utilização de CUDA <i>Streams</i> . . . . .	27
Figura 9 – Formatos de stencil comumente utilizados . . . . .	34
Figura 10 – Simulação da Equação de Laplace para uma matriz 512x512 . . . . .	35
Figura 11 – Operação de stencil do Método de Diferenças Finitas . . . . .	36
Figura 12 – Representação do ambiente rCUDA proposto . . . . .	37
Figura 13 – Comparação entre a solução numérica e a analítica da Equação de Laplace .	38
Figura 14 – Representação da matriz de diferenças entre as soluções analítica e numérica	38
Figura 15 – Discretização e particionamento do problema . . . . .	39
Figura 16 – Halos e transferências de valores entre as seções . . . . .	40
Figura 17 – Sinal recebido no receptor 19 . . . . .	41
Figura 18 – Tempos de Processamento para o Método de Eliminação Gaussiana . . . .	43
Figura 19 – Valores de <i>speedup</i> do algoritmo de Eliminação Gaussiana . . . . .	44
Figura 20 – Gráficos de nível de utilização produzidos pelo NVVP (Adaptados) . . . . .	44
Figura 21 – Valores de <i>speedup</i> CUDA e rCUDA para as aplicações de Stencil (3 pontos)	46
Figura 22 – Valores de <i>speedup</i> CUDA e rCUDA para as aplicações de Stencil (5 pontos)	46
Figura 23 – Valores de <i>speedup</i> CUDA e rCUDA para as aplicações de Stencil (9 pontos)	47
Figura 24 – Valores de <i>speedup</i> CUDA e rCUDA para o FDTD . . . . .	47
Figura 25 – Aproximações para a derivada de $f(x)$ (Adaptado de (SADIKU, 2000)) . .	56

## LISTA DE TABELAS

Tabela 1 – Especificações da Placa Gráfica Utilizada na Eliminação Gaussiana . . . . .	30
Tabela 2 – Quantidades de linhas de código acrescentadas ao algoritmo de Eliminação Gaussiana . . . . .	32
Tabela 3 – Especificações da Placa Gráfica Utilizada na Equação de Laplace . . . . .	35
Tabela 4 – Tempo médio em segundos (M) e desvio padrão (DP) dos experimentos . . .	42
Tabela 5 – Informações de desempenho para as versões CUDA e OpenACC . . . . .	43
Tabela 6 – Tempos de processamento para o stencil de 3 pontos (segundos) . . . . .	44
Tabela 7 – Tempos de processamento para o Stencil de 5 pontos (segundos) . . . . .	44
Tabela 8 – Tempos de processamento para o Stencil de 9 pontos (segundos) . . . . .	45
Tabela 9 – Tempos de processamento para o FDTD (segundos) . . . . .	45
Tabela 10 – Velocidades de transferência estimadas para o barramento local e para o rCUDA . . . . .	48

## **LISTA DE SIGLAS**

API	Application Programming Interface
CUDA	Compute Unified Device Architecture
FDM	Finite Difference Method
GCC	GNU C Compiler
MPI	Message Passing Interface
NVVP	NVIDIA Visual Profiler
rCUDA	Remote CUDA

## SUMÁRIO

<b>1</b>	<b>INTRODUÇÃO</b>	<b>13</b>
1.1	Objetivo	13
1.2	Trabalhos relacionados	14
1.3	Organização do Trabalho	15
<b>2</b>	<b>COMPUTAÇÃO PARALELA</b>	<b>16</b>
2.1	Modelos de Memória	16
2.1.1	Memória Compartilhada	16
2.1.2	Memória Distribuída	18
2.2	Sistemas Heterogêneos	20
2.3	CUDA	20
2.3.1	Arquitetura	21
2.3.2	Modelo de Programação CUDA	22
2.3.3	OpenACC	23
2.3.4	Modelo de Programação OpenACC	23
2.3.5	rCUDA	25
2.4	Avaliação de Desempenho	27
2.5	Considerações Finais	27
<b>3</b>	<b>MÉTODOS NUMÉRICOS E PROPOSTA DE AVALIAÇÃO</b>	<b>29</b>
3.1	Eliminação Gaussiana	29
3.1.1	Proposta de Metodologia de Avaliação	29
3.2	Método de Diferenças Finitas	32
3.2.1	Equação de Laplace	32
3.2.2	Stencils Computacionais	33
3.2.3	Proposta de Metodologia de Avaliação	34
3.2.4	Validação dos resultados	37
3.2.5	Decomposição do domínio	37
3.3	Método de Diferenças Finitas no Domínio do Tempo	39
3.3.1	Proposta de Metodologia de Avaliação	40
3.3.2	Decomposição do domínio	41
3.4	Considerações Finais	41
<b>4</b>	<b>RESULTADOS</b>	<b>42</b>
4.1	OpenACC	42
4.2	rCUDA	43
<b>5</b>	<b>CONCLUSÕES E TRABALHOS FUTUROS</b>	<b>49</b>
5.1	Contribuições	49
5.2	Trabalhos Futuros	50
	<b>REFERÊNCIAS</b>	<b>51</b>
	<b>APÊNDICE A – MÉTODO DA ELIMINAÇÃO GAUSSIANA</b>	<b>55</b>
A.0.1	Descrição do Método	55
	<b>APÊNDICE B – MÉTODO DA EQUAÇÃO DE LAPLACE</b>	<b>56</b>
	<b>ANEXO A – ARTIGO PUBLICADO</b>	<b>58</b>

# 1 INTRODUÇÃO

*“A ciência conhece um único comando: contribuir com a ciência.”*

Bertold Brecht

As pesquisas em Computação de Alto Desempenho vêm sendo impulsionadas ao longo do tempo a partir do avanço na produção de equipamentos com maior velocidade de processamento, e posteriormente pelo desenvolvimento de técnicas de paralelismo e passagem de mensagens, como o PVM (*Parallel Virtual Machine* - Máquina Virtual Paralela) (GEIST, 1994) e o MPI (*Message Passing Interface* - Interface de Passagem de Mensagens), tornando-se este último posteriormente um padrão *de facto* (PACHECO, 1997).

Logo esses novos recursos passaram a ser explorados em diversas outras áreas, tornando necessário o desenvolvimento de aplicações que pudessem tirar proveito dos mesmos. (DON-GARRA et al., 1995).

O aprimoramento das técnicas tradicionais, baseadas na evolução das CPUs (*Central Processing Units* - Unidade Central de Processamento) chegou ao seu limite em termos de eficiência energética (FLOPs/Watt), demandando o desenvolvimento de alternativas. A computação baseada em sistemas heterogêneos, como as GPUs (*Graphical Processing Unit* - Unidade Gráficas de Processamento) popularizou-se rapidamente, ao oferecer dispositivos de baixo consumo de energia e níveis massivos de paralelismo (HWU, 2015).

A combinação entre dispositivos heterogêneos e as técnicas de computação distribuída permitiram um novo avanço no desempenho e na escalabilidade de aplicações de Alto Desempenho.

Porém o desenvolvimento de aplicações é limitado pela complexidade envolvida no processo de compreender os pormenores das arquiteturas, de modo a adaptar as abordagens originais para seus equivalentes mais eficientes. Essa limitação torna necessário o desenvolvimento de técnicas que permitam “habilitar mais usuários a obter vantagens dessas arquiteturas sem a necessidade de conhecimento detalhado do hardware” (REYES et al., 2012).

## 1.1 Objetivo

O presente trabalho tem por objetivo avaliar o desempenho da utilização de novos recursos computacionais baseados em aceleradores gráficos, tais como as diretivas para aceleradores e o acesso remoto a GPUs, de modo que estes permitam a obtenção de ganhos de desempenho em aplicações científicas, com menor esforço de adaptação de código e a partir da utilização de equipamentos de baixo custo. Para demonstrar a viabilidade de soluções com essas características são utilizados o padrão de programação para aceleradores baseado em diretivas e uma arquitetura MultiGPU distribuída utilizando um software cliente/servidor capaz de viabilizar o uso simultâneo de múltiplas placas gráficas, abstraindo ao programador a camada de rede. Para

ambos, são apresentados os ganhos de desempenho obtidos a partir da aplicação na aceleração de métodos numéricos.

## 1.2 Trabalhos relacionados

A utilização de processadores gráficos de propósito geral em aplicações científicas tem sido o alvo de diversos trabalhos:

Uma GPU, com a utilização da tecnologia CUDA, é empregada por SOUZA et al. (2014) na aceleração do método de Otimização por Enxame de Partículas (PSO - *Particle Swarm Optimization*), apresentando como resultado diminuições significativas nos tempos de execução, bem como a produção de um código mais legível e adaptável.

XIE; WU; CHENG (2015) apresentam o uso combinado de GPUs e CPUs aplicados à análise do tráfego em redes de alta velocidade, demonstrando o aumento na vazão de processamento para o sistema apresentado. O trabalho indica que o uso combinado de CPUs e GPUs apresenta uma capacidade de gerar uma vazão de dados superior em relação às implementações de análise de tráfego baseadas apenas em CPU, viabilizando a implementação de sistemas mais eficientes de detecção de intrusão de rede.

MANFROI et al. (2014) apresenta uma avaliação de desempenho de aceleradores em ambiente virtualizado, aplicadas à Álgebra Linear Densa. As GPUs apresentaram tempos de execução muito próximos aos obtidos por meio do acesso direto pelo sistema hospedeiro, mostrando que as ferramentas de virtualização avaliadas permitem o uso satisfatório de dispositivos aceleradores para a classe de problemas avaliados.

WANG (2014) apresenta o uso de ferramentas de simulação de dinâmica de fluidos utilizando GPUs, aplicadas à avaliação das condições da aerodinâmica de aeronaves, com o objetivo de melhorar a eficiência das mesmas, reduzindo o consumo de combustível necessário à propulsão, diminuindo conseqüentemente o impacto ambiental. O uso de GPUs reduz o tempo de processamento, permitindo que a simulação utilize stencils computacionais de alta ordem, melhorando a precisão dos modelos.

Uma implementação baseada em GPUs, utilizando CUDA, foi desenvolvida por PITAKSI-RIANAN; NOURI; TU (2016) para implementar um algoritmo de análise estatística de dados científicos. Os resultados obtidos demonstraram a obtenção de ganhos de desempenho superiores em pelo menos um ordem de magnitude em relação as implementações de referência utilizando CPUs.

Experimentos multiGPU utilizando rCUDA são apresentados por REAÑO et al. (2015), desenvolvedores do framework, utilizando uma implementação do algoritmo de Montecarlo disponibilizada com a instalação do kit de desenvolvimento CUDA. Os autores demonstram que o framework rCUDA pode ser utilizado para diminuir os custos com a aquisição e o consumo de energia em clusters, ao tornar um conjunto de GPUs disponível em um nó acessível aos demais por meio da rede.

### **1.3 Organização do Trabalho**

O presente trabalho está organizado da seguinte forma:

No Capítulo 2 são apresentados os conceitos referentes à Programação Paralela, os principais aspectos da computação baseada em GPUs e o método de avaliação do desempenho.

No Capítulo 3 são expostos os métodos numéricos avaliados: A Eliminação Gaussiana e o Método das Diferenças Finitas, bem como a metodologia utilizada na Avaliação de desempenho destes.

Os resultados obtidos a partir da Avaliação de Desempenho são apresentados no Capítulo 4.

No Capítulo 5 são descritas as conclusões obtidas a partir dos experimentos e os possíveis trabalhos futuros.



## 2 COMPUTAÇÃO PARALELA

*“Por vezes sentimos que aquilo que fazemos não é senão uma gota de água no mar. Mas o mar seria menor se lhe faltasse uma gota.”*

Madre Teresa de Calcutá

O avanço científico tem produzido ao longo dos anos uma demanda crescente por poder computacional. O desenvolvimento de arquiteturas de processamento paralelo tem proporcionado avanços à computação científica, resultando em equipamentos capazes de produzir resultados em intervalos de tempo menores e com custos mais baixos.

O aumento do desempenho computacional por meio do aumento da frequência de *clock* das CPUs tradicionais passou a ser limitado rapidamente em termos físicos. A manutenção do aumento constante de desempenho passou a depender da combinação com melhores arquiteturas (HEROUX; RAGHAVAN; SIMON, 2006).

O benefício das técnicas de paralelismo para o desenvolvimento de pesquisas científicas pode ser comprovado a partir de trabalhos que demonstram melhorias significativas de desempenho obtidas por meio do uso de processamento paralelo em problemas de computação científica nas mais diversas áreas do conhecimento, como por exemplo, na biologia (MANAVSKI; VALLE, 2008), na medicina (PAN; GU; XU, 2008), na engenharia (SOUZA et al., 2011), na sísmica (ABDELKHALEK et al., 2009) e na economia (BÉNYÁSZ; CSER, 2010).

Modelos de Computação Paralela tem por objetivo oferecer APIs (*Application Programming Interface*) e demais ferramentas computacionais de modo a padronizar e simplificar a utilização dos recursos do hardware para processamento paralelo (KIRK; HWU, 2013).

No presente capítulo serão abordados os modelos de classificação e as principais características dos sistemas computacionais paralelos.

### 2.1 Modelos de Memória

Um outra classificação de arquiteturas de computação paralela refere-se ao modelo de acesso à memória. De acordo com essa classificação definem-se dois tipos básicos de arquiteturas: As baseadas em Memória Compartilhada e em Memória Distribuída.

#### 2.1.1 Memória Compartilhada

Em uma arquitetura paralela baseada em memória compartilhada, as unidades de processamento têm acesso direto ao mesmo espaço de endereçamento de memória. Sistemas desse tipo promovem facilidade na implementação de algoritmos paralelos, porém têm sua escalabilidade normalmente limitada pela capacidade de expansão dos recursos do sistema. Esta limitação

se dá primariamente pela elevação dos custos dos elementos de interconexão e coerência, devido ao aumento da complexidade destes elementos a medida que é aumentada a escala (LENOSKI; WEBER, 2014).

Os primeiros sistemas computacionais paralelos basearam-se na adição de núcleos ao processador. Tais sistemas foram originalmente empregados na computação científica e em grandes sistemas comerciais, sendo popularizados no início do século XXI (DUBOIS; ANNAVARAM; STENSTRÖM, 2012).

#### 2.1.1.1 OpenMP

Iniciado em 1997, o OpenMP consiste em uma API para processamento paralelo em ambiente de memória compartilhada, portátil para diferentes plataformas e disponível para as linguagens C/C++ e *Fortran* (CHANDRA et al., 2001). É possível distribuir o fluxo de trabalho em diversas *threads*, de modo a aumentar o nível de utilização de sistemas com múltiplos processadores ou núcleos de processamento na solução de um problema.

O modelo de programação OpenMP consiste na utilização de diretivas, delimitando o trecho de código no qual haja partes paralelizáveis. As diretivas consistem em marcações especiais, processadas na etapa inicial da compilação, e que instruem o compilador a implementar um comportamento específico.

Na linguagem C, as diretivas OpenMP são especificadas por meio dos mecanismos *pragma*, que fazem parte do padrão de implementação da linguagem. Caso o compilador não possua suporte ao OpenMP, as diretivas serão ignoradas e será produzido um programa com execução sequencial (OPENMP, 2015). Esta abordagem torna simples a adaptação de algoritmos sequenciais legados, sem que seja necessário alterar a estrutura dos mesmos.

As diretivas OpenMP podem ser classificadas em três tipos básicos:

- Estruturas de controle
- Estruturas de sincronização
- Ambientes de Dados

A execução paralela no OpenMP é baseada no padrão *fork-join*. De acordo com OPENMP (2015) um programa inicia sua execução a partir de uma *thread*, chamada de "*thread* inicial". Durante a execução de uma região paralela, ocorre o *fork*, que faz com que o trecho identificado seja executado por um conjunto de *threads*, originadas a partir da *thread* inicial. Ao final da execução paralela, cada uma das *threads* entra em processo de sincronização e é encerrada, ocorrendo a etapa de *join*, na qual o fluxo de execução retorna à *thread* inicial. A Figura 1 ilustra o padrão *fork-join*.

DIRK; ATLE; S. (2016) apresentam os resultados da avaliação de desempenho do OpenMP em sistemas de Memória Compartilhada compostos por milhares de *cores*, baseados na arquitetura Numascale.

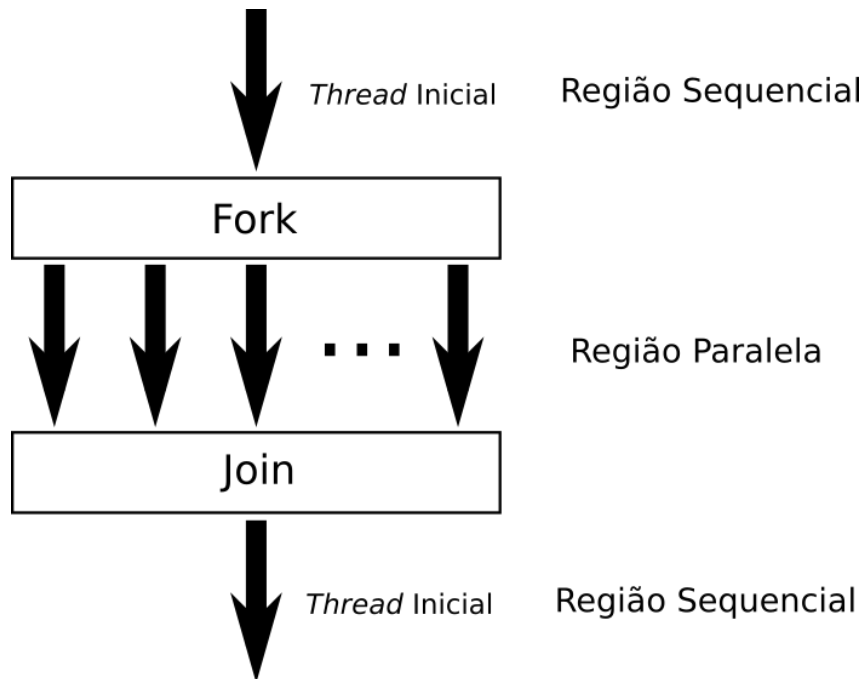


Figura 1 – Padrão *Fork-Join* utilizado no OpenMP (Fonte: Elaborada pelo autor)

### 2.1.2 Memória Distribuída

Em um sistema paralelo de memória distribuída, cada unidade de processamento possui seu próprio espaço de memória. As unidades de processamento são interligadas por uma rede e a troca de informações precisa ser feita por meio de troca de mensagens.

A possibilidade de interconectar unidades de processamento por meio de uma rede de computadores passou a ser explorada a partir da necessidade de expansão de recursos computacionais para o processamento de problemas maiores.

A utilização de sistemas computacionais de Memória distribuída tornou-se popular devido a relativa facilidade de implementação, por meio da utilização de computadores e conexões de rede padrão. Sistemas interconectados por meio de uma rede dedicada são comumente chamados de *clusters*. Esses sistemas são normalmente programáveis por meio de uma interface de passagem de mensagens RAUBER; RÜNGER (2013). A Figura 2 apresenta o modelo genérico de um sistema computacional baseado em Memória Distribuída.

#### 2.1.2.1 MPI

O MPI é talvez a interface de mensagens mais amplamente conhecida (COOK, 2013). É amplamente utilizado em ambientes de computação científica, aplicado ao uso em clusters computacionais.

O MPI consiste em um padrão de interface para passagem de mensagens em programação paralela. O processo de padronização foi iniciado na conferência de Supercomputing, de 1992, e

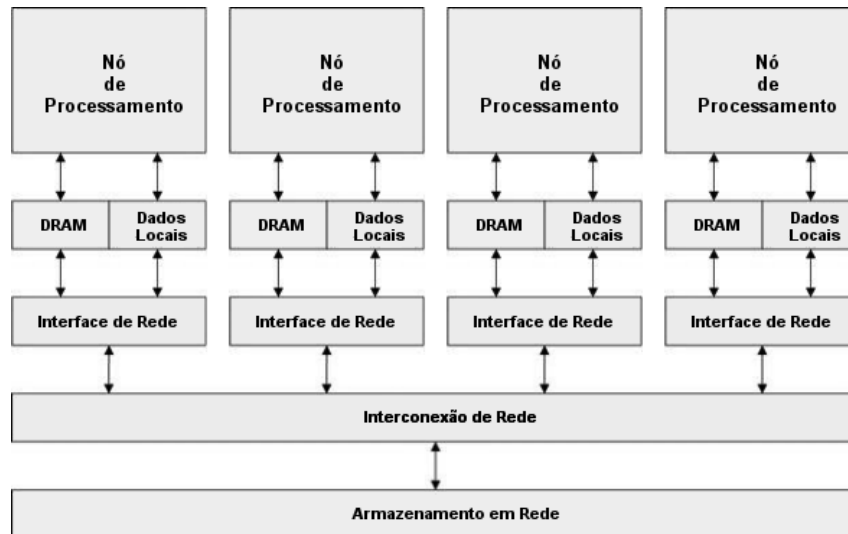


Figura 2 – Representação de um sistema em Memória Distribuída (adaptado de (COOK, 2013))

teve a participação de diversos fabricantes de equipamentos, universidades e centros de pesquisa GROPP et al. (2014). O padrão especifica os seguintes aspectos:

- Comunicação Ponto-a-ponto,
- Tipos de dados,
- Operações Coletivas,
- Grupos de processos,
- contextos de comunicação,
- topologias de processo,
- Gerenciamento do ambiente e investigação,
- O objeto Info,
- criação e gerenciamento de processos,
- comunicação unilateral,
- Interfaces externas,
- Entrada e Saída de arquivos paralela,
- Componentes para linguagens Fortran e C,
- Suporte a Ferramentas.

O MPI apenas define as interfaces necessárias, mas não as implementa. Existem diversas implementações do padrão, dentre as quais destacam-se as implementações livres MPICH, LAM/MPI e OpenMPI (RAUBER; RÜNGER, 2013). Existem ainda implementações derivadas como o Intel MPI e o Tianhe MPI, utilizado no Supercomputador Tianhe-2<sup>1</sup>, ambas implementações são baseadas no MPICH.

<sup>1</sup>O Tianhe-2 foi o líder da lista To500 entre 2013 e 2015. A lista Top500 é mantida pela universidade de Mannheim, desde 1993 e é utilizada como referência de capacidade computacional pela indústria mundial de supercomputadores.

## 2.2 Sistemas Heterogêneos

Segundo HWU (2015), as Arquiteturas de Sistema Heterogêneas consistem em uma nova plataforma de hardware e camada de software associada, que habilita processadores de diferentes tipos a trabalharem associativa e cooperativamente juntos em memória compartilhada. As GPUs se enquadram nesse tipo de arquitetura, sendo estas as precursoras a demandar o desenvolvimento de técnicas eficientes para o uso de co-processadores (HWU, 2015).

Um sistema heterogêneo CPU/GPU é uma solução que se beneficia das características complementares entre os dois tipos de processador, tirando o melhor proveito de ambos em prol do aumento da performance (CHENG; GROSSMAN; MCKERCHER, 2014). Para tanto, é necessário identificar no código-fonte as partes do programa apropriadas à execução em GPU, mantendo a execução em CPU das demais partes. Essa é a base dos modelos de programação CUDA e OpenACC, apresentados na Seção 2.3. A Figura 3 apresenta a representação esquemática básica de um sistema heterogêneo CPU/GPU.

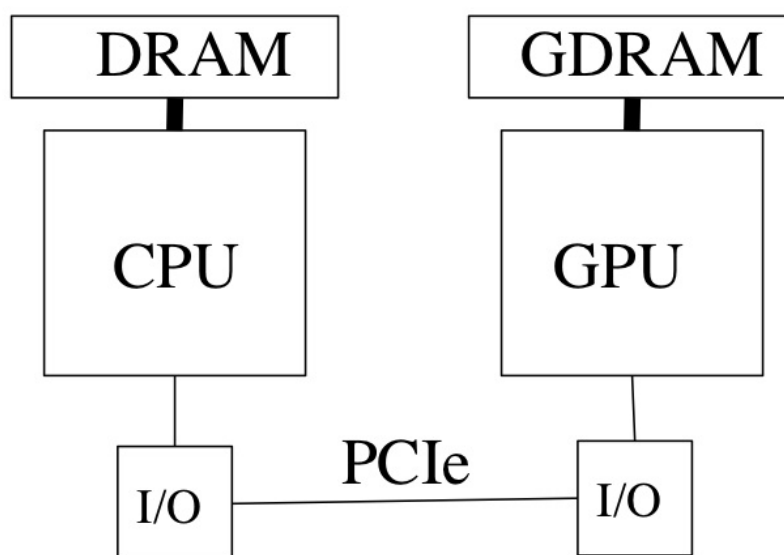


Figura 3 – Arquitetura básica de um sistema heterogêneo CPU/GPU (Fonte: Elaborada pelo autor)

## 2.3 CUDA

Criada pela empresa de placas gráficas NVIDIA, a *Compute Unified Device Architecture* (CUDA) fornece uma plataforma de computação paralela que permite uma programação, em processadores gráficos, menos complexa, por meio de um conjunto de instruções próprias para a arquitetura GPGPU (*General Purpose Graphics Processing Unit*).

Disponibilizada para as linguagens C/C++ e *Fortran*, permite um alto nível de paralelismo, já que foi projetada para obter ganhos de desempenho por meio do aumento da vazão (*throughput*) de processamento (KIRK; HWU, 2013). No entanto, apresenta, talvez, como única restri-

ção, o fato do conjunto necessário para o desenvolvimento utilizando a plataforma, consistindo em uma placa gráfica NVIDIA compatível, o driver de dispositivo e o conjunto de ferramentas CUDA *Toolkit*, que provê, entre outros itens, os compiladores e bibliotecas específicos (SANDERS; KANDROT, 2010), serem fornecidos apenas pela própria NVIDIA.

### 2.3.1 Arquitetura

Em termos lógicos, CUDA apresenta as *threads* agrupadas em *Blocks*. Os *Blocks*, por sua vez, são organizados em *Grids*, desse modo, é necessário, ao executar um *kernel* CUDA, estabelecer quantos *Blocks* de *threads* irão executar a mesma operação, e quantas *threads* comporão cada *Block*. Apenas *threads* de um mesmo *Block* são capazes de trocar informações em memória compartilhada entre si. Os limites de tamanho dos *Blocks* e *Grids* são estabelecidos de acordo com as especificações de hardware de cada placa, conforme ilustrado na Figura 4.

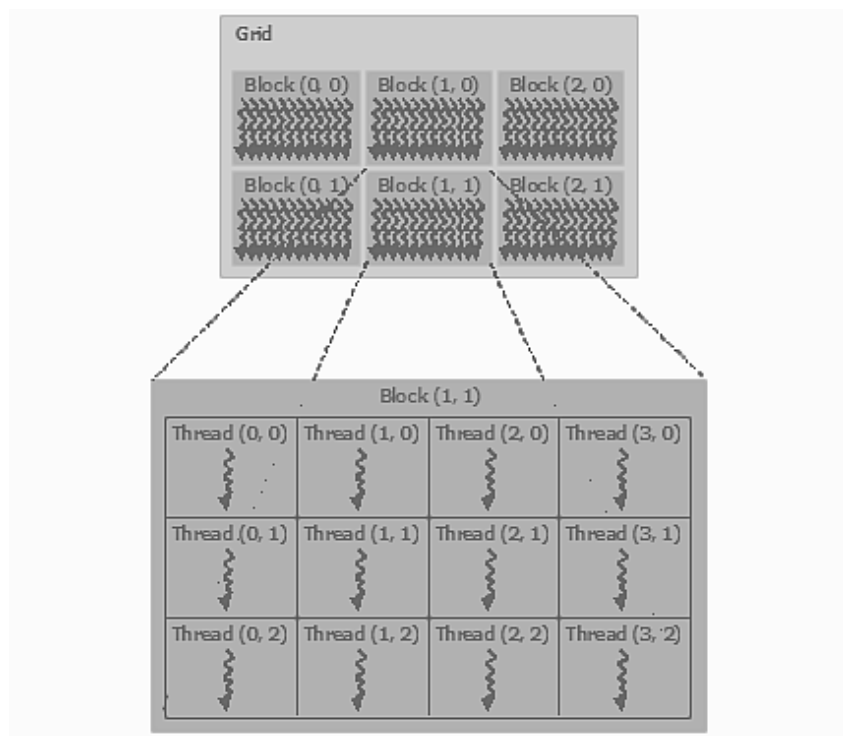


Figura 4 – Arquitetura CUDA (Fonte: (NVIDIA, 2013))

Fisicamente, cada GPU é composta por um conjunto de *Streaming Multiprocessors* (SM ou SMX). Cada SM dispõe de um espaço de memória de alta velocidade de acesso, que pode ser compartilhada dentro dos *Blocks* da *Grid* entre as *threads* que compõem o respectivo *Block*. Uma representação esquemática da arquitetura de um sistema GPU/CPU é apresentado na Figura 5, na qual são apresentados os SMs, as estruturas de cache, e a possível conexão de múltiplas GPUs em um mesmo *host*, por meio de barramento, o que permite estabelecer uma analogia arquitetural com os sistemas de memória distribuída baseados em CPU (vide Figura 2), uma vez que também apresenta unidades compostas por um grupo de núcleos que compartilham um

espaço de memória (GPUs), sendo estas unidades interconectadas, no caso de um *host* multiGPU, por um barramento, normalmente PCI-e, ou ainda por meio de uma camada de rede, como na arquitetura apresentada neste trabalho.

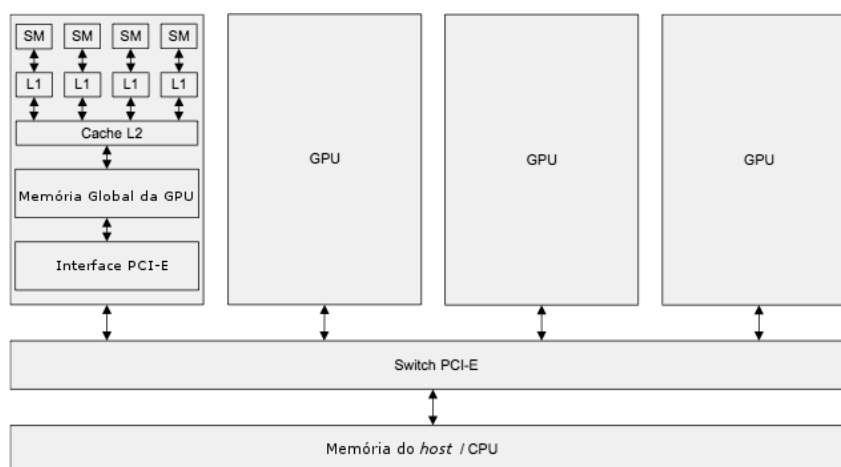


Figura 5 – Representação de uma GPGPU (adaptado de (COOK, 2013))

### 2.3.2 Modelo de Programação CUDA

A estrutura básica de um programa CUDA consiste na inicialização dos espaços de memória na GPU, seguida da transferência de dados da memória do *host* para os espaços de memória inicializados na placa gráfica. Os trechos de código que correspondem ao paradigma SPMD (*Single Program Multiple Data*) são implementados em uma função denominada *kernel*. Após a cópia dos dados, a operação definida no *kernel* é aplicada em simultâneo a uma quantidade massiva de dados por meio de múltiplas *threads*, o que permite a redução do tempo de processamento. Ao final do processamento, os dados referentes aos resultados são transferidos para a memória principal do sistema *host* (RAUBER; RÜNGER, 2013).

Um exemplo simples de código escrito utilizando CUDA C é mostrado no Código 2.1.

```

1  #include <stdio.h>
2  #define N 10000
3  __global__ void Vector_Add(int *dev_a, int *dev_b, int *dev_c) {
4      int tid = threadIdx.x + blockIdx.x * blockDim.x;
5      if(tid < N)
6          dev_c [tid] = dev_a[tid] + dev_b[tid];
7  }
8  int main (void) {
9      int Host_a[N], Host_b[N], Host_c[N];
10     int *dev_a , *dev_b, *dev_c;
11     cudaMalloc((void **)&dev_a , N*sizeof(int));
12     cudaMalloc((void **)&dev_b , N*sizeof(int));
13     cudaMalloc((void **)&dev_c , N*sizeof(int));
14     for (int i = 0; i<N; i++){

```

```

15     Host_a[i] = -i;
16     Host_b[i] = i*i;
17 }
18 cudaMemcpy(dev_a, Host_a, N*sizeof(int), cudaMemcpyHostToDevice);
19 cudaMemcpy(dev_b, Host_b, N*sizeof(int), cudaMemcpyHostToDevice);
20 Vector_Add<<<128,128>>> (dev_a, dev_b, dev_c);
21 cudaMemcpy(Host_c, dev_c, N*sizeof(int) , cudaMemcpyDeviceToHost);
22 for(int i = 0; i<N; i++)
23     printf("%d+%d=%d\n", Host_a[i], Host_b[i], Host_c[i]);
24 cudaFree(dev_a);cudaFree(dev_b);cudaFree(dev_c);
25 return 0;
26 }

```

Código 2.1 – Exemplo de código CUDA C

Nas linhas 3 a 7 é definido o *kernel* a ser executado na GPU. As linhas 9 e 10, apresentam a definição das variáveis na *host* e no dispositivo. O espaço na memória do dispositivo é alocado para as variáveis nas linhas 11 a 13. Nas linhas 18 e 19, os valores dos vetores *Host\_a* e *Host\_b* são copiados para os vetores correspondentes no dispositivo (*dev\_a* e *dev\_b*). A chamada ao *kernel* é feita na linha 20. Na linha 21 os valores do vetor *dev\_c* são copiados para a memória do *host*, para o vetor *Host\_c*.

### 2.3.3 OpenACC

Fundado por um consórcio entre as empresas NVIDIA, PGI, CRAY e CAPS Enterprise, o OpenACC implementa um conjunto de diretivas de compilação para as linguagens C/C++ e *Fortran* que permite a criação de códigos capazes de tirar proveito de aceleradores (como os Processadores Gráficos de Propósito Geral) por meio da identificação dos laços com diretivas de compilação, semelhantes às utilizadas pelo modelo OpenMP (OPENACC, 2014). Futuramente, as diretivas OpenACC deverão passar a integrar as especificações do modelo OpenMP (OPENACC, 2014), permitindo que programadores possam fazer uso dos aceleradores disponíveis com pouco esforço, abstraindo os pormenores da comunicação do processador principal com os mesmos.

O OpenACC torna-se adequado a situações nas quais os programadores não estão familiarizados com o modelo de programação CUDA ou estão satisfeitos em abstrair os detalhes da arquitetura-alvo (COOK, 2013).

### 2.3.4 Modelo de Programação OpenACC

O uso do OpenACC consiste na identificação de regiões de maior carga computacional, marcando-as com as diretivas correspondentes, para que possam ser executadas no acelerador. Caso um compilador compatível não seja utilizado, ou se não houver uma *flag* de compilação



correspondente, código será compilado de forma a produzir um código sequencial. Adicionalmente, é possível identificar nas diretivas as variáveis contendo os dados a serem transferidos entre a memória do sistema *host* e a acelerador.

#### 2.3.4.1 Gangs, Workers e Vectors

Do ponto de vista do programador trabalhando com estruturas *parallel* ou *kernel*, o modelo de execução OpenACC tem três níveis: *Gangs*, *Workers* e *Vectors*. A maneira como essas construções são mapeadas para o hardware subjacente depende dos recursos do dispositivo e o que o compilador supõe que é o melhor mapeamento para o problema. (FARBER, 2016)

As diretivas utilizadas são apresentados nos tópicos a seguir, de acordo com a documentação apresentada em (OPENACC, 2014).

#### 2.3.4.2 Diretiva Kernels

Permite identificar regiões que possam ser automaticamente convertidas em kernels, para execução no dispositivo acelerador. Um exemplo da diretiva é apresentado no Código 2.2.

```

1  #pragma acc kernels
2  {
3    for (i = 0; i < SIZE; ++i) {
4      for (j = 0; j < SIZE; ++j) {
5        for (k = 0; k < SIZE; ++k) {
6          c[i][j] += a[i][k] * b[k][j];
7        }
8      }
9    }
10 }
```

Código 2.2 – Exemplo de uso da diretiva Kernels

#### 2.3.4.3 Diretiva loop

Identifica especificamente um laço, gerando um código correspondente otimizado no acelerador. Em caso de loops aninhados, é possível utilizar o modificador *independent* para otimizar execução de loops independentes. O Código 2.3 apresenta um exemplo de uso da diretiva.

```

1  #pragma acc loop
2  for( int i = 0; i < n; ++i )
3    y[i] += a*x[i];
```

Código 2.3 – Exemplo de uso da diretiva Loop

### 2.3.4.4 Diretivas Data

Usadas para definir explicitamente a transferência de dados entre o *host* e o dispositivo acelerador. Embora a identificação da demanda pela cópia dos dados seja feita de forma automática, essa diretiva permite demarcar esta cópia de forma a reduzir transferências desnecessárias. O Código 2.4 exemplifica o uso da diretiva `data`.

```

1 #pragma acc data copyin(a,b) copy(c)
2 {
3   ...
4 }
```

Código 2.4 – Exemplo de uso da diretiva `Data`

### 2.3.5 rCUDA

Desenvolvido por um grupo de pesquisa da *Universitat Politecnica de Valencia*, o rCUDA consiste em um *framework* para virtualização remota de GPUs. A solução permite que o acesso remoto a placas gráficas compatíveis com CUDA seja feito de forma transparente à aplicação.

A ferramenta funciona por meio de uma arquitetura cliente/servidor, conforme apresentado na Figura 6. No cliente, as bibliotecas de execução do CUDA são substituídas pelas versões rCUDA por meio de bibliotecas *wrapper*, que recebe as requisições a recursos da GPU e envia a um servidor, por meio de uma API de *sockets*. O servidor correspondente a cada placa gráfica virtual é definido por meio de variáveis de ambiente. O servidor rCUDA executa um *daemon*, que entrega as requisições à GPU e responde ao cliente.

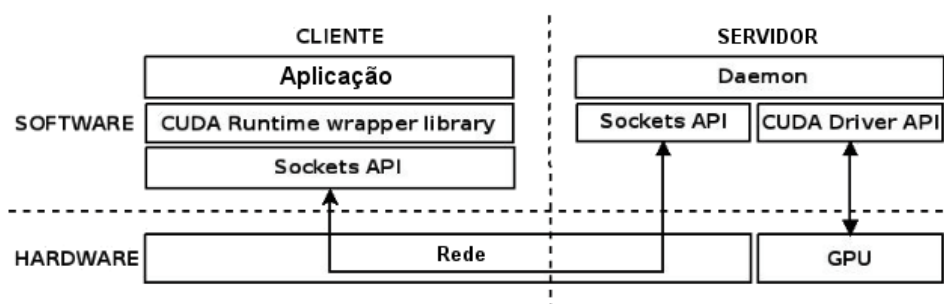


Figura 6 – Diagrama da arquitetura rCUDA (Adaptado de (RCUDA, 2016))

Por meio do rCUDA é possível executar códigos multiGPU associando as GPUs de múltiplas máquinas a um único cliente, contornando a limitação de uma única placa comumente encontrada em equipamentos de baixo custo. A solução suporta a utilização de redes baseadas nas arquiteturas TCP/IP e Infiniband®.

A proposta original da aplicação é disponibilizar GPUs CUDA de forma concorrente entre dispositivos, desacoplando-as dos *hosts* nos quais estão instaladas. Segundo RCUDA (2016), o objetivo do *framework* é prover três tipos de cenários:

- Aumentar a taxa de utilização dos nós dotados de GPUs em *Clusters e Datacenters*
- Oferecer acesso compartilhado de poucas GPUs de alto desempenho, disponibilizando-as a equipamentos de baixo custo em laboratórios acadêmicos
- Habilitar o acesso a GPUs a partir de máquinas virtuais

O diagrama na Figura 7 apresenta a representação simplificada de um exemplo de ambiente rCUDA. É possível observar os clientes conectados ao servidor rCUDA por meio de uma rede, e este disponibilizando a ambos o acesso virtual à GPU disponível.

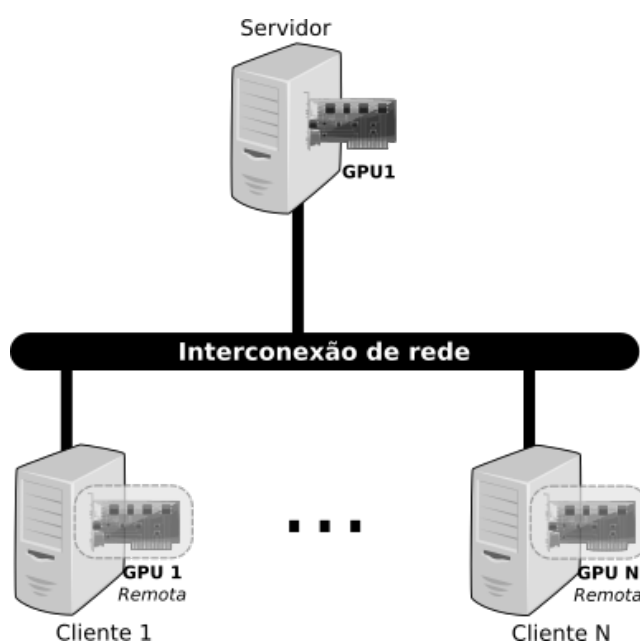


Figura 7 – Representação de um ambiente rCUDA

### 2.3.5.1 CUDA Streams

Para que a execução do processamento possa ocorrer simultaneamente em múltiplas placas, é necessário iniciar fluxos de processamento paralelos. A API CUDA prevê essa possibilidade por meio do uso de *CUDA Streams*.

De acordo com SANDERS; KANDROT (2010), um *CUDA Stream* representa uma fila de operações a serem executadas em uma ordem específica, permitindo que operações como o lançamento de *kernels* e cópias de dados em memória possam se executadas em paralelo, ao serem executadas em *Streams* diferentes.

Quando não referenciada explicitamente, cada chamada de *kernel* ou transferência de dados é feita a partir da *stream* padrão (*stream0*). Desse modo a próxima operação na GPU é iniciada apenas após o término da anterior. Ao instanciar diferentes *streams* é possível atribuir diferentes tarefas a estas, criando fluxos assíncronos de execução. A Figura 8 ilustra o processo de funcionamento de *streams*.

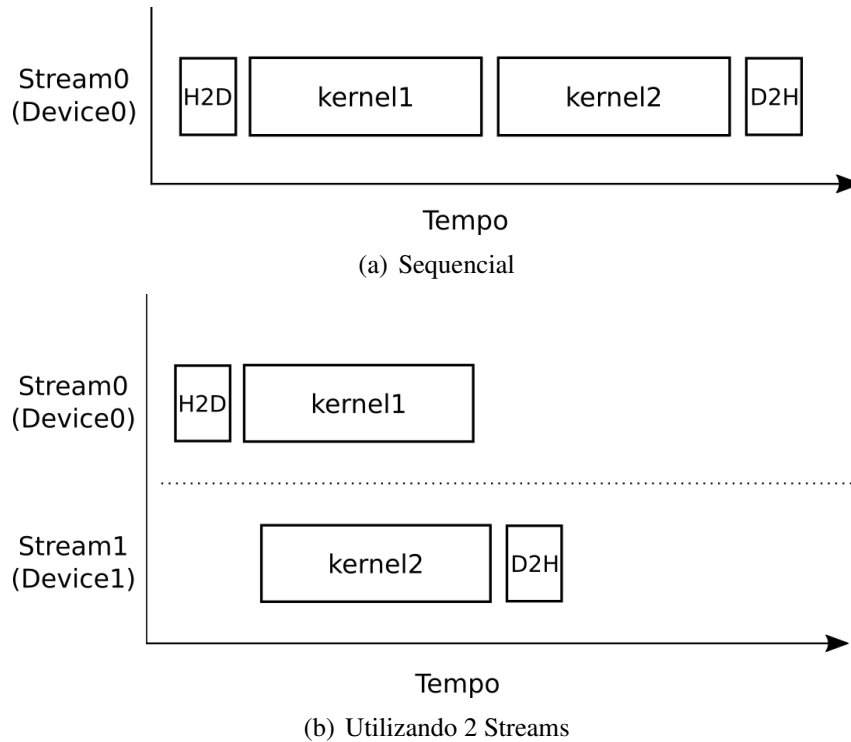


Figura 8 – Utilização de CUDA Streams

## 2.4 Avaliação de Desempenho

O *speedup* é uma das métricas utilizadas na avaliação do desempenho de sistemas paralelos, sendo definido por JAMSHED (2015) como a razão do tempo de processamento necessário para completar determinado cálculo utilizando um único processador quando comparado com o cálculo equivalente sendo executado por uma máquina de processamento concorrente.

O *speedup* pode ser expresso na Equação (2.1), onde  $S_n$  refere-se ao *speedup* para  $n$  processadores,  $T_s$  refere-se ao tempo de execução serial, e  $T_n$  ao tempo de execução utilizando  $n$  processadores.

$$S_n = \frac{T_s}{T_n} \quad (2.1)$$

A obtenção de uma série de valores de *speedup* permite analisar a escalabilidade de uma aplicação, de acordo com a variação de um determinado parâmetro, como o tamanho do problema.

## 2.5 Considerações Finais

Neste capítulo foram apresentados os principais aspectos relacionados ao uso de técnicas de paralelismo aplicável a computação científica, destacando-se o uso de aceleradores gráficos no incremento do desempenho desta categoria de aplicações. Foram apresentados os modelos de

programação e as arquiteturas de *hardware* e rede utilizadas nas implementações avaliadas.

### 3 MÉTODOS NUMÉRICOS E PROPOSTA DE AVALIAÇÃO

*“Os números são as regras dos seres e a Matemática é o Regulamento do Mundo.”*

F. Gomes Teixeira

Os Métodos Numéricos permitem definir problemas matemáticos em termos de equações, de modo que sejam solucionados por meio de operações aritméticas. (CHAPRA; CANALE, 2016).

Neste capítulo são apresentados os métodos de Eliminação Gaussiana, aplicado a solução de matrizes densas, o FDM (*Finite Difference Method* - Método das Diferenças Finitas), aplicado à solução da Equação de Laplace, e o método FDTD (*Finite-Difference Time-Domain* - Diferenças Finitas no Domínio do Tempo), utilizado em uma aplicação do Eletromagnetismo.

Com o objetivo de indicar a viabilidade das ferramentas apresentadas como alternativas para a aceleração de aplicações científicas, são apresentadas duas propostas de avaliação. Na seção 3.1.1 será apresentada a metodologia utilizada para avaliar o desempenho do uso do OpenACC aplicado a aceleração do algoritmo de Eliminação Gaussiana. Os métodos utilizados na avaliação de desempenho do uso da arquitetura MultiGPU baseada em rCUDA serão apresentados na seção 3.2.3.

#### 3.1 Eliminação Gaussiana

Sistemas de Equações Lineares podem ser definidos como conjuntos de equações de primeiro grau que possuem as mesmas incógnitas (BARBOSA, 2011). Com base nesta característica é possível organizar o sistema na forma de uma matriz de coeficientes que multiplicada por um vetor de incógnitas resulta no vetor de termos independentes. Resolver um sistema linear corresponde a encontrar os valores dos elementos do vetor de incógnitas para o qual as equações sejam verdadeiras. A descrição detalhada do método da Eliminação Gaussiana é apresentada no Apêndice A.

##### 3.1.1 Proposta de Metodologia de Avaliação

Os testes realizados consistiram na adaptação do trecho de maior intensidade computacional, correspondendo à implementação do método da Eliminação Gaussiana. Os códigos foram escritos na linguagem C e posteriormente incluídas as diretivas de cada ferramenta, para a execução dos comparativos. Os resultados obtidos foram coletados com a interface gráfica do Sistema Operacional desabilitada (em modo texto).

As especificações da Placa Gráfica NVIDIA M2075 são apresentadas na Tabela 1:

Tabela 1 – Especificações da Placa Gráfica Utilizada na Eliminação Gaussiana

<b>Especificação</b>	<b>Valor</b>
Multiprocessadores	14
CUDA Cores	448
GPU Clock	1147 MHz
Quantidade de memória Global	6144 MB
CUDA Capability	2.0

Foi utilizada uma rotina para gerar matrizes de acordo com a ordem fornecida como argumento e os sistemas lineares correspondentes foram solucionados. O programa foi executado de maneira sequencial por dez vezes. O tempo médio de execução foi obtido, servindo como referência para o cálculo de desempenho (*speedup*) das versões paralelas. O trecho principal da versão serial do código é apresentado no Código 3.1:

```

1  for (i=0; i<(n-1); i++)
2      for (j=(i+1); j<n; j++)
3          ratio = A[j][i] / A[i][i];
4          for (k=i; k<n; k++)
5              A[j][k] -= (ratio * A[i][k]);
6          b[j] -= (ratio * b[i]);

```

Código 3.1 – Trecho do Código da Versão Serial

Na primeira etapa de paralelização do código sequencial foram adicionadas as diretivas OpenMP ao código, sendo este executado utilizando todos os núcleos das CPUs disponíveis, perfazendo um total de trinta e dois núcleos. Este passo foi repetido por dez vezes para a obtenção do tempo médio de execução. No Código 3.2 é apresentado o trecho principal do código, acrescido das informações pragmáticas do OpenMP.

```

1  #pragma omp parallel shared(A,b,n,i) private(j,k)
2
3  for (i=0; i<(n-1); i++)
4  #pragma omp parallel for
5      for (j=(i+1); j<n; j++)
6          ratio = A[j][i] / A[i][i];
7          for (k=i; k<n; k++)
8              A[j][k] -= (ratio * A[i][k]);
9          b[j] -= (ratio * b[i]);

```

Código 3.2 – Trecho do Código da Versão em OpenMP

Na segunda etapa de paralelização, foram adicionadas as diretivas OpenACC, compatíveis com o código anteriormente modificado para execução em OpenMP, por meio de variáveis específicas definidas no ambiente de execução, pode-se comprovar a utilização dos processadores gráficos na execução paralela dos laços. O algoritmo foi novamente executado por dez vezes, para obtenção do tempo médio de execução. O trecho principal acrescido das informações pragmáticas do OpenACC é mostrado no Código 3.3.

```

1  #pragma acc data copy(A[:n][:n],b[0:n])
2
3  for(i=0;i<(n-1);i++)
4  #pragma acc region
5  #pragma acc loop independent
6      for(j=(i+1);j<n;j++)
7          ratio = A[j][i] / A[i][i];
8  #pragma acc loop independent
9      for(k=i;k<n;k++)
10         A[j][k] -= (ratio * A[i][k]);
11         b[j] -= (ratio * b[i]);

```

Código 3.3 – Trecho do Código na Versão em OpenACC

Na terceira etapa, o código foi parcialmente reescrito, para tornar-se compatível com o CUDA C, com a inserção de um *kernel* e de funções responsáveis pela troca de dados com os processadores gráficos. O código em CUDA C foi otimizado, por meio do uso de estruturas de memória compartilhada de acesso rápido (*Shared Memory*). A utilização da *Shared Memory* é possível entre *threads* de um mesmo bloco. Os valores com maior quantidade de acessos, correspondentes à linha de referência para cada iteração, foram carregados para a *Shared Memory*, aumentando a velocidade de acesso. O código reescrito foi então compilado com o compilador C da NVIDIA, disponível no *CUDA Toolkit*, e executado por dez vezes para a obtenção do tempo médio de execução. O trecho principal do código, referente ao *kernel* CUDA produzido, é apresentado no Código 3.4.

```

1  __global__ void factor(double *a_d, int size, long i)
2  {
3      long j = blockIdx.y*blockDim.y+threadIdx.y;
4      long k = blockIdx.x*blockDim.x+threadIdx.x;
5
6      __shared__ double lpivo[BLOCKSIZE+1];
7
8      if(threadIdx.y==0)
9          lpivo[threadIdx.x] = a_d[i*(size+1)+k];
10     __syncthreads();
11
12     double pivo = a_d[i*(size+1)+i];
13     if(j<(size) && k<(size+1)){
14         const double ratio = a_d[j*(size+1)+i]/pivo;
15         if(j>=(i+1) && k>=(i)){
16             a_d[j*(size+1)+k] = a_d[j*(size+1)+k] - (ratio*lpivo[threadIdx.x]);
17         }
18     }
19 }

```

Código 3.4 – Trecho do *Kernel* em CUDA



Outro fator analisado, neste trabalho, foi a quantidade de linhas necessárias para a adaptação do código-fonte original. Tais quantidades de linhas inseridas são apresentadas na Tabela 2. A implementação em CUDA demandou um acréscimo maior na quantidade de linhas pois requer que as funções para alocação de dados na memória da GPU, bem como para transferências de dados entre as memórias sejam explicitadas no código. Além disso, é necessária a definição de uma função separada (kernel) a ser invocada para execução na GPU. Tais estruturas não se fizeram necessárias nas versões baseadas em diretivas, ou puderam ser inseridas de forma mais concisa.

Tabela 2 – Quantidades de linhas de código acrescentadas ao algoritmo de Eliminação Gaussiana

<b>Implementação</b>	<b>Número de linhas adicionadas</b>
OpenMP	4
OpenACC	6
CUDA	28

### 3.2 Método de Diferenças Finitas

Determinar o padrão de comportamento de determinado fenômeno, de modo a ser capaz de prevê-lo, é o objetivo em diversas áreas das ciências naturais. No entanto, a formulação analítica de alguns fenômenos demonstra-se inviável ou mesmo impraticável por conta da complexidade. Em tais casos, o uso de ferramentas matemáticas permite a análise do fenômeno de interesse com um grau de aproximação que satisfaça a aplicação correspondente. Dentre o conjunto de técnicas alternativas destaca-se Método de Diferenças Finitas, que permitem substituir equações diferenciais por equações de diferenças finitas (SADIKU, 2000). Uma descrição mais detalhada do método é apresentada no Apêndice B.

#### 3.2.1 Equação de Laplace

A equação de Laplace é um caso particular de Equação Diferencial Parcial com diversas aplicações na Física, como no eletromagnetismo, na mecânica de fluidos e na condução de calor, uma vez que descreve o comportamento de campos quando dissipados em meios homogêneos. Em sua formulação tridimensional, a Equação de Laplace pode ser expressa conforme mostrado na Equação (3.1).

$$\nabla^2 V(x, y, z) = \frac{\partial^2}{\partial x^2} V(x, y, z) + \frac{\partial^2}{\partial y^2} V(x, y, z) + \frac{\partial^2}{\partial z^2} V(x, y, z) = 0 \quad (3.1)$$

A Equação (3.2) corresponde à Equação de Laplace Bidimensional.

$$\nabla^2 V(x, y) = \frac{\partial^2}{\partial x^2} V(x, y) + \frac{\partial^2}{\partial y^2} V(x, y) = 0 \quad (3.2)$$

Considerando-se a discretização do domínio em uma malha de nós, será utilizada a notação  $v(i, j)$  para referenciar o valor de uma função  $v(x, y)$  nas coordenadas  $x = x_i$  e  $y = y_j$ . Escrevendo a aproximação numérica para as derivadas de segunda ordem (equações B.5 e B.6), a equação de Laplace 2D é aproximada na forma

$$\frac{v(i+1, j) - 2v(i, j) + v(i-1, j)}{(\Delta x)^2} + \frac{v(i, j+1) - 2v(i, j) + v(i, j-1)}{(\Delta y)^2} = 0 \quad (3.3)$$

No caso das discretizações nas duas direções serem iguais ( $\Delta x = \Delta y$ ), a equação é simplificada para

$$v(i+1, j) - 2v(i, j) + v(i-1, j) + v(i, j+1) - 2v(i, j) + v(i, j-1) = 0, \quad (3.4)$$

o que nos permite obter

$$v(i, j) = \frac{v(i+1, j) + v(i-1, j) + v(i, j+1) + v(i, j-1)}{4} \quad (3.5)$$

A implementação computacional do Método da Diferenças Finitas consiste na aplicação de stencils, que serão descritos na Seção 3.2.2.

### 3.2.2 Stencils Computacionais

Stencils computacionais podem ser definidos como operadores aplicados sobre cada um dos elementos de uma matriz, de modo a atualizar seu valor baseando-se nos valores dos elementos vizinhos, utilizando um padrão fixo (YANG; GUO, 2005). A operação é normalmente repetida em sucessivos instantes de tempo, a partir de uma condição inicial definida em todos os elementos da matriz, de modo a simular a difusão ou outro processo físico ao longo do tempo (RAHMAN, 2013).

Operações de stencil são utilizadas em diversas aplicações da computação científica. Exemplos incluem algoritmos empregados no processamento de imagens, nos quais as células são *pixels*; métodos numéricos utilizados para calcular soluções aproximadas de equações diferenciais parciais sobre um domínio físico, nos quais as células de caso são pequenas regiões contíguas do domínio e simulações de autômatos celulares complexos, das quais Jogo da Vida de Conway é um exemplo simples bastante conhecido (CASANOVA; LEGRAND; ROBERT, 2008).

Motivada pela importância dessa classe de problemas e sua aplicabilidade, desde de 2014 é promovido o HiStencils (Workshop Internacional em Computação de Stencils de Alto Desempenho) (HISTENCILS, 2016), durante a Conferência anual da HiPEAC. (WANG et al., 2015)

Em algumas das aplicações citadas, podem ocorrer cenários nos quais o domínio é grande o suficiente para extrapolar os limites de memória compartilhada disponível, tornando necessária uma implementação em memória distribuída.

Em termos de paralelismo, o *loop* de iterações de aplicação do stencil não é paralelizável, uma vez que ocorre dependência de valores entre as iterações, porém, o cálculo de cada elemento é independente dentro da mesma iteração, permitindo um elevado nível de paralelismo, o que o torna um algoritmo apropriado à execução em GPU. A complexidade computacional de uma operação de stencil pode ser expressa como  $\mathcal{O}(k \cdot N^2)$  (CECILIA; GARCÍA; UJALDÓN, 2012). A Figura 9 apresenta graficamente três formatos de stencil comumente utilizados em aplicações científicas: Um stencil de 3 pontos, 9(a); o stencil de Von Neuman, com 5 pontos, 9(b); e o stencil de Moore, com 9 pontos, 9(c).

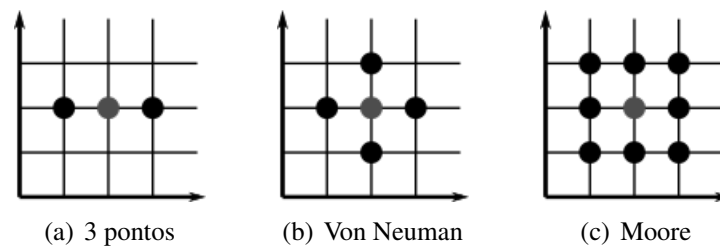


Figura 9 – Formatos de stencil comumente utilizados

O Algoritmo 1 representa os passos básicos da implementação de uma operação de um stencil de 5 pontos (mostrado na Figura 11), sobre uma matriz  $v$ , de  $i \times j$  elementos, por  $t$  instantes de tempo.

---

**Algoritmo 1** Operação básica de um stencil.

---

```

1:  $A \leftarrow \langle \text{valorinicial} \rangle$ 
2: for  $t \leftarrow 0, \langle \text{tempofinal} \rangle$  do
3:   for  $j \leftarrow 0, j$  do
4:     for  $i \leftarrow 0, i$  do
5:        $v[i, j] \leftarrow \frac{v[i+1, j] + v[i-1, j] + v[i, j+1] + v[i, j-1]}{4}$ 
6:     end for
7:   end for
8: end for

```

---

### 3.2.3 Proposta de Metodologia de Avaliação

Com o objetivo de avaliar o desempenho do uso do rCUDA em uma arquitetura para programação multiGPU, foi definido um ambiente de testes composto por três computadores *desktop* de configuração idêntica:

- 1 Processador Intel® Core i5 3.10GHz
- 8 GB de memória RAM
- 1 Placa Gráfica NVIDIA® GeForce GT 630

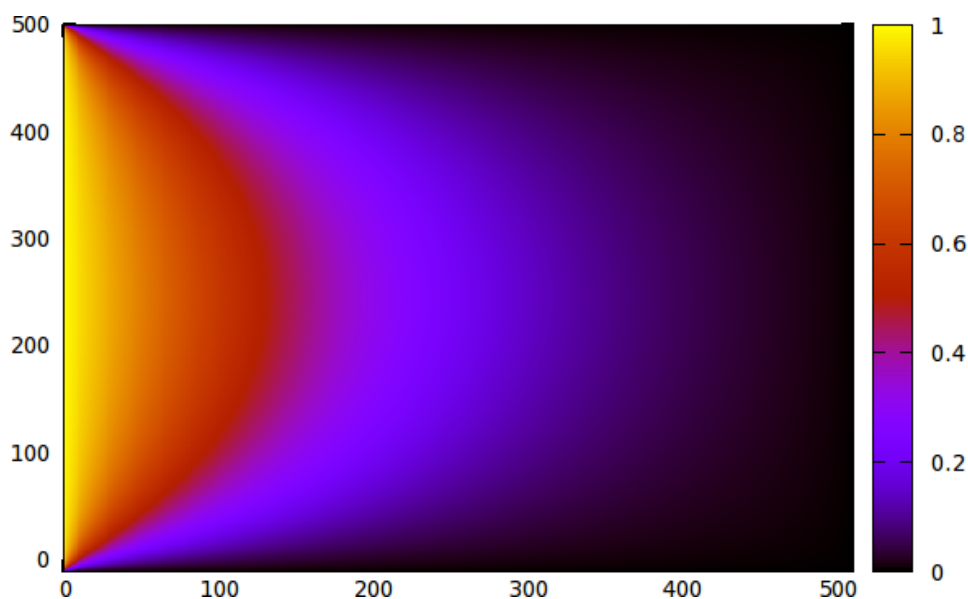


Figura 10 – Simulação da Equação de Laplace para uma matriz 512x512

- Sistema Operacional Ubuntu Linux 14.04 (x86\_64)

Os resultados obtidos foram coletados com a interface gráfica do Sistema Operacional desabilitada (em modo texto).

As especificações da Placa Gráfica NVIDIA GT630 são apresentadas na Tabela 3:

Tabela 3 – Especificações da Placa Gráfica Utilizada na Equação de Laplace

<b>Especificação</b>	<b>Valor</b>
Multiprocessadores	2
CUDA Cores	96
GPU Clock	1400 MHz
Quantidade de memória Global	4096 MB
CUDA Capability	2.1

A representação gráfica da simulação de evolução do potencial elétrico, obtida a partir da implementação utilizada no presente trabalho, é apresentada pela Figura 10. A figura representa, por meio de uma escala de cores, os valores de uma matriz que corresponde a uma região discretizada em uma malha de pontos. O valor inicial do potencial elétrico em cada um dos pontos interno é igual a 0. O valor do potencial elétrico na borda esquerda é fixo e igual a 1, enquanto o valor nas demais bordas também é fixo, porém igual a 0, caracterizando uma condição de contorno de Dirichlet (SALSA, 2015). O gráfico obtido mostra a evolução dos valores de campo após 10000 iterações.

Foi elaborado um algoritmo que aplica um stencil sobre uma matriz quadrada (correspondente à discretização do problema), repetindo-o por 10000 iterações. A ordem da matriz quadrada variou entre os valores 1024, 2048, 3072, 4096 e 5120, a fim de observar o comportamento da aplicação a medida em que é aumentado o tamanho do problema.

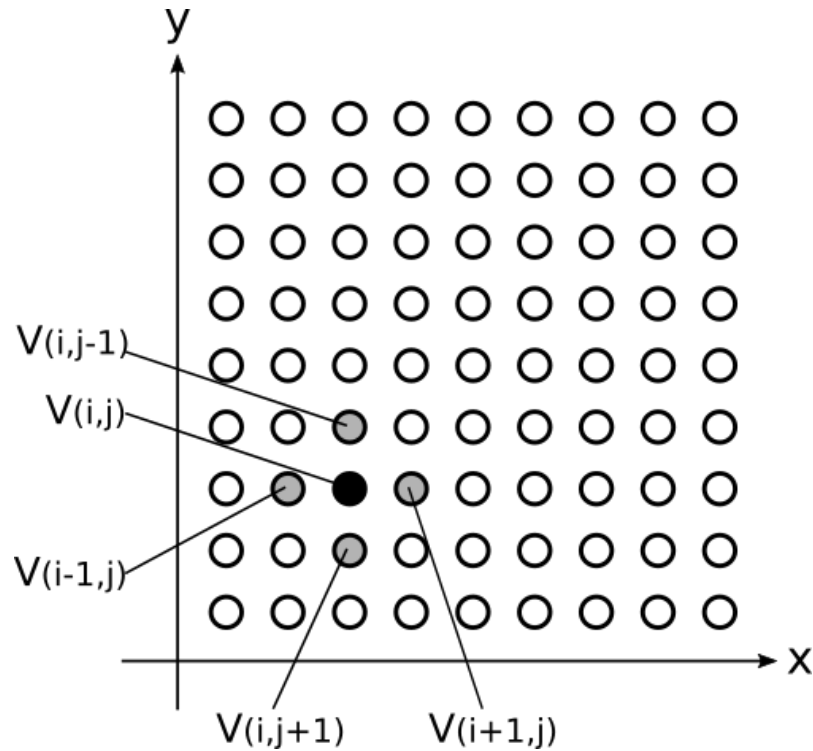


Figura 11 – Operação de stencil do Método de Diferenças Finitas

Com o objetivo de ampliar a avaliação de desempenho, de modo a avaliar a variação deste em função da intensidade computacional, foram utilizados três formatos de stencil: Um com 3, outro com 5 e outro com 9 pontos, correspondentes a outras ordens de aproximação da solução numérica.

Cada um dos valores da matriz corresponde a um dos pontos  $V(i, j)$ . A cada iteração, cada valor de  $V(i, j)$  é atualizado a partir dos valores vizinhos  $V(i-1, j)$ ,  $V(i+1, j)$ ,  $V(i, j-1)$  e  $V(i, j+1)$ , conforme ilustrado pela Figura 11.

A aplicação foi inicialmente executada em sua versão sequencial, de modo a obter valores de referência para o cálculo de *speedup* (fator de aceleração). Cada um dos stencils foi executado 10 vezes para cada uma das ordens de matrizes definidas, obtendo-se os valores de média e desvio-padrão.

Posteriormente a aplicação foi adaptada para ser executada em CUDA. Foi escrito um *kernel* simples, que implementa a aplicação dos stencils. A aplicação foi executada também por 10 vezes para cada um dos formatos de stencil, correspondentes a diferentes ordens de aproximação, sobre cada um dos tamanhos de matrizes já mencionados, obtendo-se a média e o desvio-padrão.

Foram feitas as configurações iniciais em uma rede Gigabit Ethernet e o *daemon* do rCUDA foi iniciado nos computadores designados a disponibilizar as GPUs. Algumas variáveis de ambiente foram definidas no computador designado a acessar as GPUs remotas, desabilitando o acesso à GPU local. Como resultado, foi obtida uma arquitetura multiGPU baseada nas GPUs remotas disponibilizadas. Como procedimento de verificação, foi executado o programa *devi-*

*ceQuery*, que detecta GPUs CUDA. O programa identificou a disponibilidade dos dispositivos compatíveis. A Figura 12 apresenta um diagrama com a representação da arquitetura avaliada.

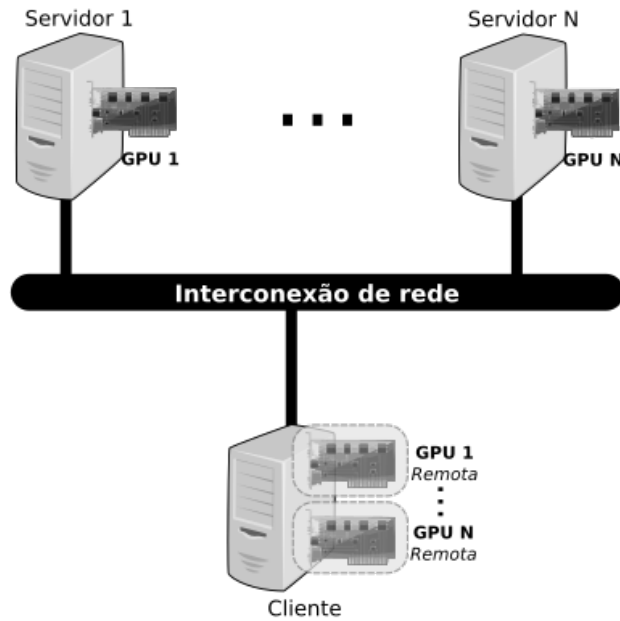


Figura 12 – Representação do ambiente rCUDA proposto

### 3.2.4 Validação dos resultados

Para efeito de validação dos resultados, os valores obtidos foram comparados com a implementação baseada na solução analítica do problema. A implementação consistiu na solução por meio da série trigonométrica apresentada por GRIFFITHS (2013).

$$V(x, y) = \sum_{n=1}^{\infty} \frac{4v(t)}{\pi} \frac{1}{n} \frac{\sinh(\frac{nx}{a}y)}{\sinh(\frac{nxb}{a})} \sin \frac{nk}{a}x \quad (3.6)$$

aplicada a cada um dos pontos da matriz de mesma dimensão.

A solução foi implementada no *software* Matlab™, utilizando uma aproximação para 100 termos da série.

Os valores obtidos mostraram-se compatíveis com a implementação numérica. As imagens correspondentes à solução analítica e à numérica são apresentadas na Figura 13.

As figuras apresentam o mesmo resultado, excetuando-se os erros de arredondamento. Uma representação gráfica da matriz de diferenças entre as imagens é mostrada na Figura 14..

### 3.2.5 Decomposição do domínio

O código CUDA foi alterado novamente, para utilizar uma abordagem multiGPU. Foram adicionados os procedimentos para dividir a matriz em seções, enviando-as para as GPUs cor-

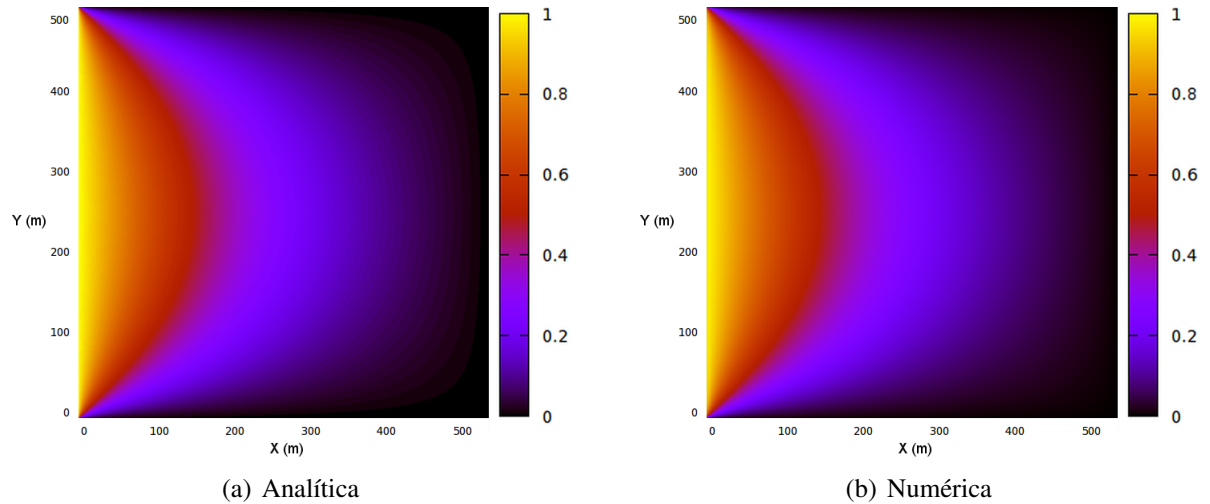


Figura 13 – Comparação entre a solução numérica e a analítica da Equação de Laplace

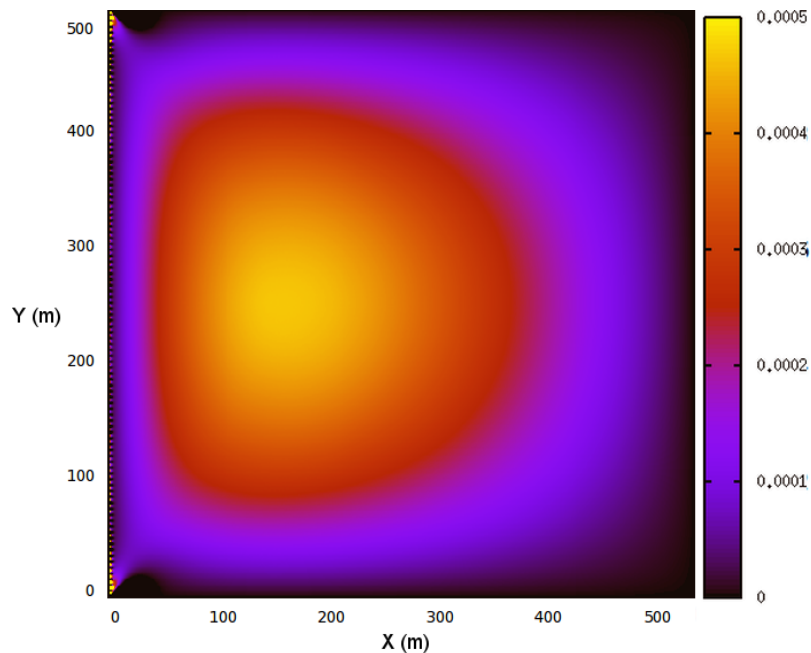


Figura 14 – Representação da matriz de diferenças entre as soluções analítica e numérica

respondentes, por meio da função `cudaSetDevice()`. A Figura 15 ilustra o processo divisão do problema. A matriz original, de ordem  $N$ , foi subdividida equitativamente em seções horizontais  $S$ , correspondendo a blocos de linhas, sendo  $n$  de 1 a  $N$ .

Uma vez que a atualização do valor de cada elemento  $V(i, j)$  é baseada nos valores de elementos adjacentes, o cálculo dos valores próximos às bordas das seções depende de valores pertencentes à seção adjacente. Desse modo faz-se necessária a transferência desses valores entre as GPUs.

A Figura 16 apresenta uma representação das transferências de valores entre as seções do problema. As linhas correspondentes às bordas de cada seção foram duplicadas (representadas em tom mais escuro) para armazenar os valores vizinhos, criando regiões comumente chama-

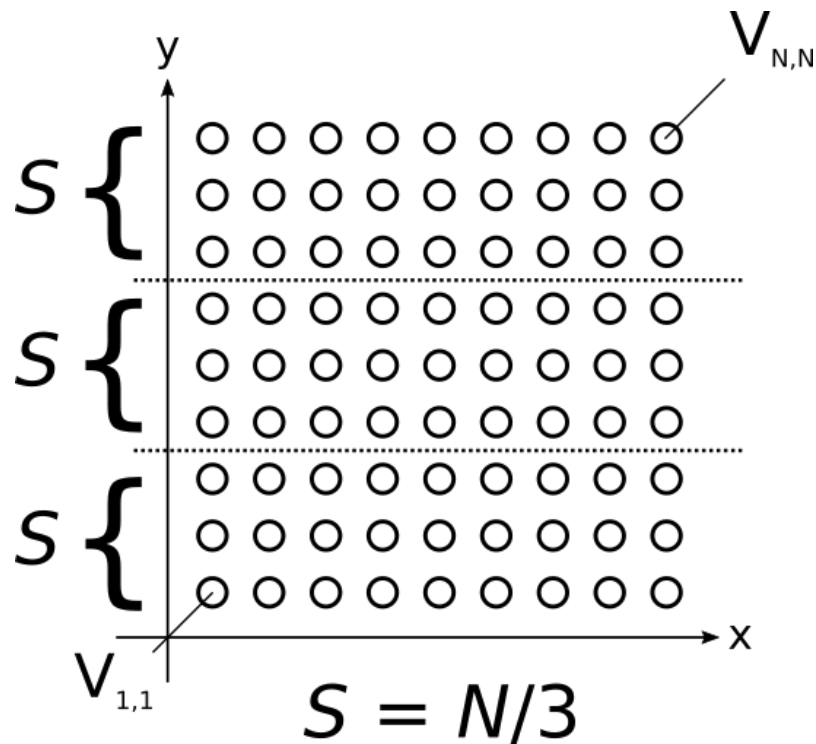


Figura 15 – Discretização e particionamento do problema

das de "halos". Após cada iteração, os valores destas regiões, correspondentes aos valores de potencial elétrico nas linhas de vizinhança entre as seções, são atualizados (conforme indicado pelas setas), de modo a serem utilizados na atualização correta dos valores nas linhas de borda, garantindo a correta continuidade na atualização dos valores.

Foi adicionada uma função para atualização de bordas, com o objetivo de enviar à parte seguinte os valores atualizados das bordas da parte anterior. Uma vez que a versão do rCUDA utilizada não implementa comunicação do tipo *DeviceToDevice*, foi necessário, a cada iteração, transferir os valores das bordas para a memória do *host* e posteriormente desta para a memória do dispositivo de destino correspondente.

O código foi novamente executado por 10 vezes para os formatos e ordens definidos anteriormente, obtendo-se os valores de média e desvio-padrão. Foram medidos os tempos de processamento utilizando duas e três GPUs.

### 3.3 Método de Diferenças Finitas no Domínio do Tempo

Introduzida por YEE (1966), a técnica de Diferenças Finitas no Domínio do Tempo (Finite-Difference Time-Domain - FDTD) consiste em um método numérico para a solução discreta das Equações de Maxwell por meio de diferenças centradas, aplicável à simulação da propagação de ondas eletromagnéticas em meios isotrópicos.

O método destaca-se pela relativa simplicidade de implementação e eficiência na obtenção de valores em uma ampla faixa de frequência. Em razão de tais características, o método pas-



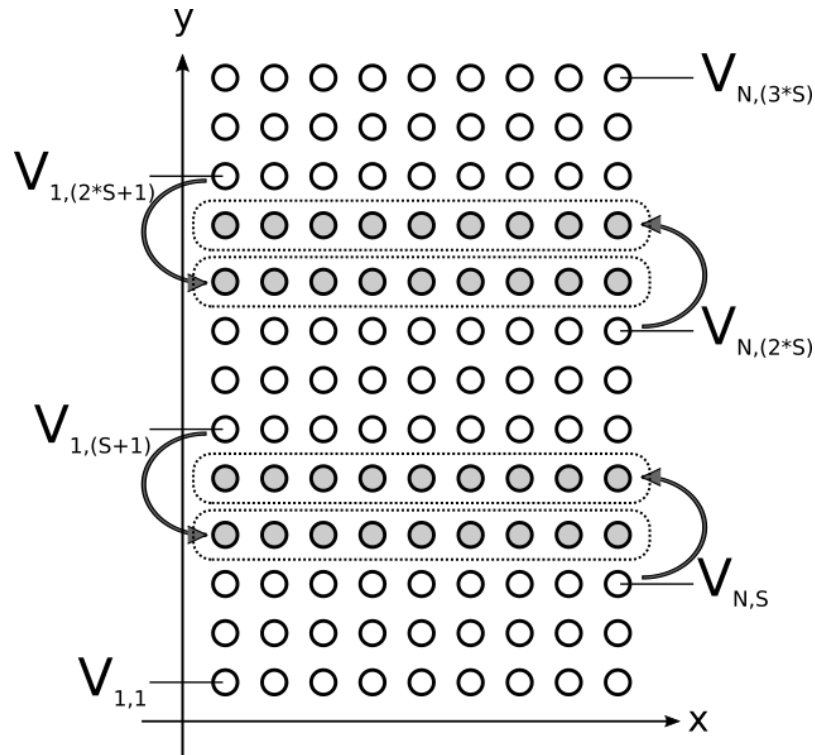


Figura 16 – Halos e transferências de valores entre as seções

sou a ser utilizado em uma quantidade crescente de aplicações no eletromagnetismo, a medida maiores capacidades de processamento tornaram-se disponíveis.

### 3.3.1 Proposta de Metodologia de Avaliação

Com o objetivo de avaliar de maneira consistente o desempenho obtido na arquitetura multiGPU apresentada, foi executada sobre a mesma uma aplicação baseada no método FDTD.

A aplicação, desenvolvida pelo Laboratório de Eletromagnetismo da Universidade Federal do Pará, consiste em um radar multiestático utilizando ondas UWB (Ultra Wide Band - banda ultralarga). O FDTD é utilizado na simulação da propagação de um pulso UWB produzido por uma fonte pontual, que é detectado por um conjunto de receptores, de modo que a variação no sinal recebido permite calcular a posição do objeto que a originou. O domínio do problema é delimitado por uma condição de contorno do tipo UPML conforme descrito por ARAUJO (2010). A aplicação implementa a propagação de onda em duas dimensões em modo Transverso Magnético ( $TM_z$ ). A representação do sinal obtido por um dos receptores na referida aplicação é apresentado na Figura 17.

Foram utilizados os mesmos equipamentos descritos na Seção 3.2.3. A implementação foi adaptada para efetuar o processamento em GPU utilizando CUDA, e posteriormente para executar em múltiplas GPUs utilizando o *framework* rCUDA. A aplicação foi executada para problemas bidimensionais com as ordens 2048, 4096, 6144 e 8192, executando 4000 iterações.

### 3.3.2 Decomposição do domínio

O domínio do problema foi decomposto de maneira análoga ao que foi apresentado na Seção 3.2.5. Foram criadas subseções das matrizes correspondentes às componentes de campo elétrico e magnético, bem como regiões de "halos" para armazenamento dos valores correspondentes às regiões vizinhas. Foram adicionadas funções para a atualização dos "halos". Após o cálculo dos valores de  $E_z$ , os valores correspondentes às regiões de vizinhança são atualizados na região de "halo" da GPU posterior. Analogamente, após o cálculo dos valores de  $H_x$  e  $H_y$ , os valores de vizinhança são transferidos para a região de "halo" da GPU anterior.

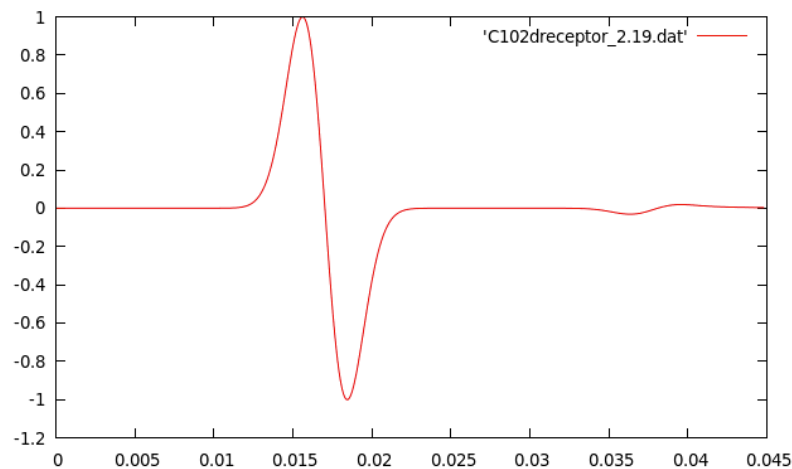


Figura 17 – Sinal recebido no receptor 19

## 3.4 Considerações Finais

No presente capítulo foram apresentados os métodos numéricos utilizados na Avaliação de Desempenho proposta neste trabalho. Os métodos selecionados foram utilizados por serem aplicáveis a cenários reais na computação científica e por permitirem implementações computacionais que destacam as características dos Aceleradores Gráficos utilizados.

Foram utilizadas dimensões de matrizes que pudessem ser armazenadas na memória da GPU, tornando possível a obtenção dos valores de *speedup* em relação às versões sequenciais.

## 4 RESULTADOS

*“Aquilo a que chamamos acaso não é, não pode deixar de ser, senão a causa ignorada de um efeito conhecido.”*

Voltaire

Neste capítulo são apresentados os resultados obtidos a partir das propostas de avaliação de desempenho do uso dos aceleradores gráficos aplicados aos algoritmos de Eliminação Gaussiana e da Equação de Laplace, bem com à uma aplicação baseada no Método FDTD.

### 4.1 OpenACC

Os tempos de processamento coletados nos testes, utilizando o método de Eliminação Gaussiana, nas versões OpenMP, OpenACC e CUDA, para as matrizes de cada uma das ordens fornecidas, são expressos na Tabela 4 e no gráfico da Figura 18.

Tabela 4 – Tempo médio em segundos (M) e desvio padrão (DP) dos experimentos

	2000		4000		6000		8000		10000	
	M	DP	M	DP	M	DP	M	DP	M	DP
<b>Serial</b>	13,65	0,02	110,34	0,28	376,28	5,24	889,69	11,84	1737,37	22,70
<b>OpenMP</b>	2,00	0,01	13,71	0,03	44,06	0,07	101,66	0,11	208,22	0,74
<b>OpenACC</b>	1,60	0,01	6,12	0,03	17,90	0,11	41,21	0,08	81,28	0,09
<b>CUDA</b>	1,12	0,03	3,65	0,03	9,79	0,04	20,93	0,04	39,04	0,08

Foram obtidos valores de *speedup* superiores a 44 em relação à versão sequencial para as matrizes de maior ordem. As médias dos valores de *speedup* obtidos para diferentes ordens de matrizes são apresentadas pelo gráfico da Figura 19.

Finalmente, foram observadas as execuções dos programas baseados em OpenACC e CUDA por meio da ferramenta NVIDIA Visual Profiler (NVVP). A ferramenta, que integra o CUDA Toolkit, permite analisar o desempenho de aplicações CUDA por meio do fornecimento de informações sobre a execução das mesmas, tais como alocação de memória e chamadas a *kernels*, apresentando-as graficamente. Os gráficos de nível de utilização das versões OpenACC e CUDA, para uma matriz de ordem 10000, são apresentados no NVIDIA Visual Profiler pela Figura 20.

A análise das informações de execução fornecidas pelo Profiler indica que a versão OpenACC, apesar de obter um melhor aproveitamento da transferência de memória, produziu um executável com menor nível de aproveitamento do paralelismo. A menor quantidade de memória compartilhada utilizada revela que a versão OpenACC teve uma perda de desempenho relacionada ao tempo de acesso à memória, uma vez que o tempo de acesso à memória global é mais elevado.

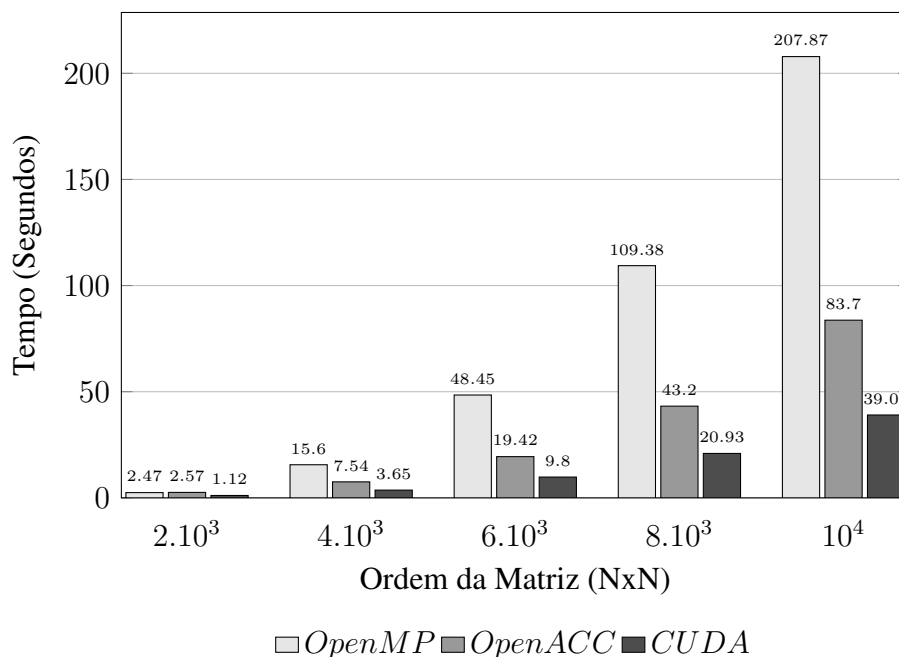


Figura 18 – Tempos de Processamento para o Método de Eliminação Gaussiana

Tabela 5 – Informações de desempenho para as versões CUDA e OpenACC

	OpenACC	CUDA
<b>Tamanho do Bloco (x,y,z)</b>	128,1,1	16,16,1
<b>Tempo total de execução do kernel</b>	80s	46s
<b>Ocupância</b>	74%	90%
<b>Quantidade de Shared Memory (por bloco)</b>	8B	4kB
<b>Vazão de armazenamento em memória global</b>	38GB/s	7,6GB/s
<b>Vazão de carregamento em memória global</b>	84,6GB/s	22,8GB/s
<b>Quantidade de acessos à memória compartilhada</b>	36980	421675250
<b>Vazão de armazenamento em memória compartilhada</b>	117GB/s	60,8GB/s
<b>Vazão de carregamento em memória compartilhada</b>	468GB/s	548GB/s

A ferramenta de *profile* da NVIDIA fornece ainda outros dados relevantes à análise do desempenho, tais como tamanhos de *Grid* e de *Blocks* utilizados, ocupância da GPU (nível de aproveitamento do paralelismo), quantidade de memória compartilhada utilizada, e vazão média de dados. Tais valores são apresentados na Tabela 5, relativa a execução das versões CUDA e OpenACC.

## 4.2 rCUDA

Os tempos de processamento coletados nos testes com stencil de 3 pontos são apresentados na Tabela 6.

A Tabela 7 apresenta os tempos de processamento obtidos na execução do stencil de 5 pontos.

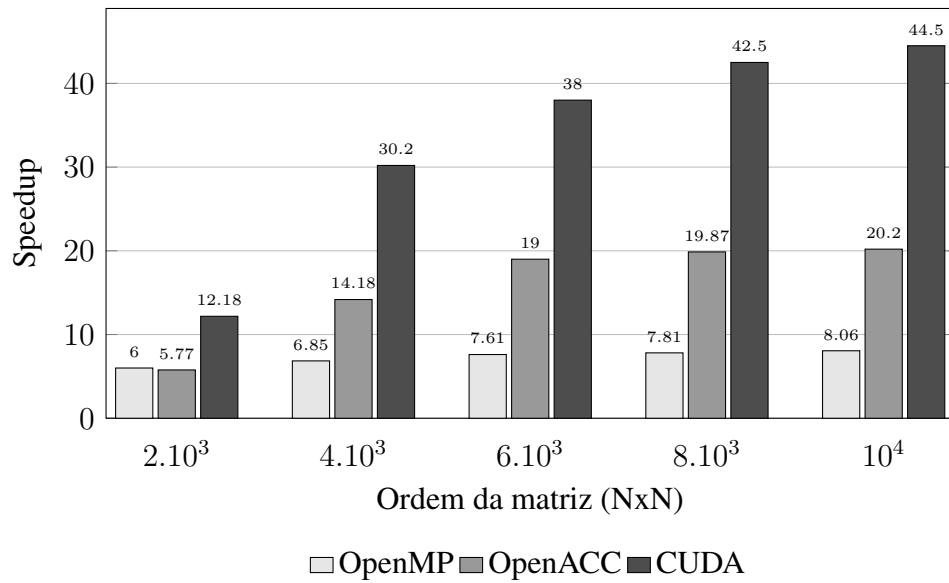


Figura 19 – Valores de *speedup* do algoritmo de Eliminação Gaussiana

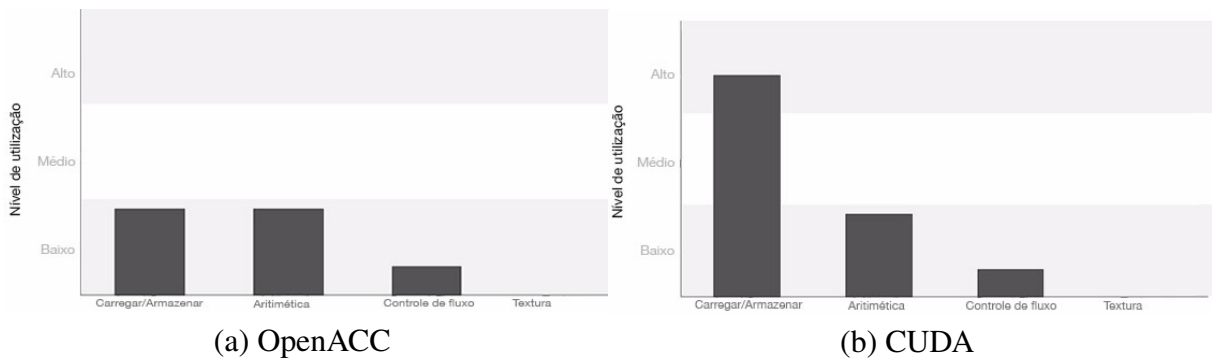


Figura 20 – Gráficos de nível de utilização produzidos pelo NVVP (Adaptados)

Tabela 6 – Tempos de processamento para o stencil de 3 pontos (segundos)

	1024		2048		3072		4096		5120	
	M	DP	M	DP	M	DP	M	DP	M	DP
<b>Serial</b>	5,69	0,02	22,83	0,04	59,53	0,04	91,54	0,04	165,81	0,03
<b>Serial O3</b>	2,04	0,01	8,16	0,01	25,95	0,04	36,54	0,01	72,11	0,04
<b>CUDA</b>	1,01	0,01	3,89	0,02	9,30	0,02	15,59	0,02	25,65	0,02
<b>rCUDA 2GPUs</b>	0,57	0,01	2,15	0,01	5,05	0,01	8,50	0,02	13,95	0,04
<b>rCUDA 3GPUs</b>	0,56	0,01	2,16	0,01	4,82	0,01	8,4	0,04	13,2	0,03

Tabela 7 – Tempos de processamento para o Stencil de 5 pontos (segundos)

	1024		2048		3072		4096		5120	
	M	DP	M	DP	M	DP	M	DP	M	DP
<b>Serial</b>	9,15	0,03	36,83	0,05	92,11	0,03	146,75	0,05	255,50	0,06
<b>Serial O3</b>	8,73	0,07	34,63	0,01	78,05	0,02	138,95	0,02	217,96	0,06
<b>CUDA</b>	1,33	0,01	5,26	0,02	12,66	0,02	21,35	0,03	35,19	0,01
<b>rCUDA 2GPUs</b>	0,77	0,01	2,99	0,01	7,08	0,02	11,98	0,03	19,60	0,04
<b>rCUDA 3GPUs</b>	0,67	0,01	2,51	0,01	5,47	0,02	9,53	0,05	16,01	0,05

Na Tabela 8 são mostrados os tempos de processamento obtidos na execução do stencil de 9 pontos.

Tabela 8 – Tempos de processamento para o Stencil de 9 pontos (segundos)

	1024		2048		3072		4096		5120	
	M	DP	M	DP	M	DP	M	DP	M	DP
<b>Serial</b>	13,85	0,01	55,75	0,07	142,99	0,08	222,58	0,05	396,17	0,06
<b>Serial O3</b>	10,55	0,01	42,31	0,03	96,02	0,03	170,28	0,04	267,94	0,01
<b>CUDA</b>	2,08	0,01	8,26	0,02	19,42	0,01	33,25	0,02	53,94	0,03
<b>rCUDA 2GPUs</b>	1,15	0,01	4,48	0,01	10,46	0,03	17,94	0,02	28,97	0,04
<b>rCUDA 3GPUs</b>	0,93	0,01	3,52	0,01	8,01	0,02	13,53	0,03	22,71	0,04

Os tempos de processamento obtidos na execução do FDTD são mostrados Na Tabela 9.

Tabela 9 – Tempos de processamento para o FDTD (segundos)

	2048		4096		6144		8192	
	M	DP	M	DP	M	DP	M	DP
<b>Serial O3</b>	432,35	0,02	1706,51	0,04	3752,03	0,03	6710,14	0,04
<b>CUDA</b>	44,81	0,01	169,86	0,02	396,52	0,02	704,41	0,03
<b>rCUDA 2GPUs</b>	12,78	0,01	48,38	0,02	102,23	0,03	183,39	0,02
<b>rCUDA 3GPUs</b>	-	0,02	23,76	0,01	49,28	0,02	84,04	0,05

Nas Tabelas 6, 7, 8 e 9, as colunas “M” e “DP”, referenciam, respectivamente o valor médio e o desvio padrão obtidos para cada conjunto de resultados. As séries de dados identificadas como “Serial O3” referem-se aos resultados obtidos a partir do código serial compilado utilizando as otimizações disponíveis para a *flag* O3 do compilador GCC e para o processador utilizado.

O valores de *speedup* correspondentes, obtidos para os stencils de 3, 5 e 9 pontos, em relação às versões seriais otimizadas, são respectivamente apresentados por meio dos gráficos da Figura 21, da Figura 22 e da Figura 23.

O valores de *speedup* correspondentes, obtidos para o algoritmo de FDTD, em relação às versões seriais otimizadas, são respectivamente apresentados por meio dos gráficos da Figura 24.

A partir dos resultados obtidos, foi feita uma avaliação adicional com o objetivo de investigar a influência da velocidade da rede no desempenho geral dos experimentos com a biblioteca rCUDA, uma vez que esta não implementa recursos de *profiling*. Desse modo, foi escrita uma simples aplicação, que apenas executa sucessivas transferências entre as memórias do *host* e do *device*.

Os tempos de execução obtidos a partir dessa aplicação permitiram estimar as velocidades de transferência obtidas através do barramento PCI-e local e da rede Gigabit Ethernet utilizada pelo rCUDA. Os resultados obtidos são apresentados na Tabela 10:

A análise das informações permite observar um ganho constante de desempenho das versões CUDA a medida em que o tamanho do problema é aumentado.

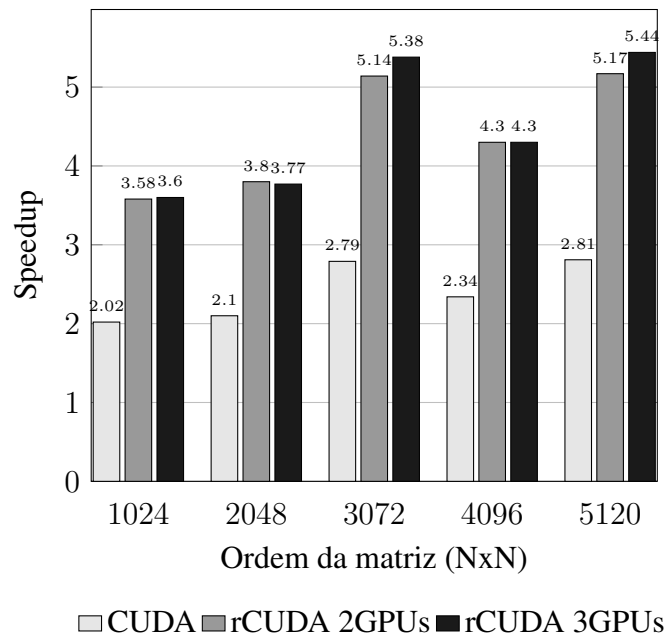


Figura 21 – Valores de *speedup* CUDA e rCUDA para as aplicações de Stencil (3 pontos)

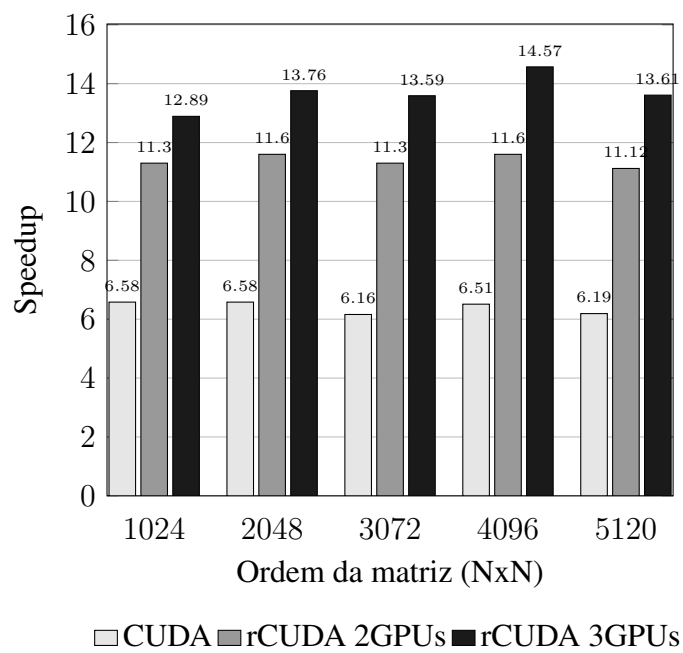


Figura 22 – Valores de *speedup* CUDA e rCUDA para as aplicações de Stencil (5 pontos)

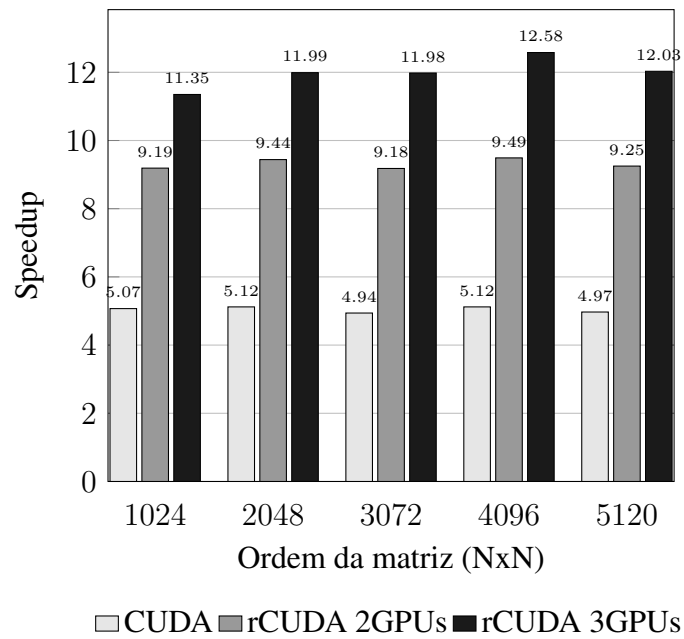


Figura 23 – Valores de *speedup* CUDA e rCUDA para as aplicações de Stencil (9 pontos)

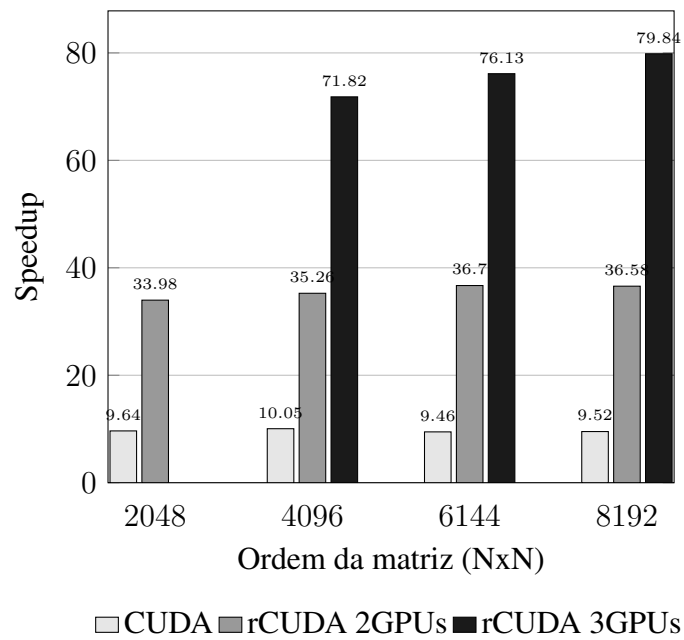


Figura 24 – Valores de *speedup* CUDA e rCUDA para o FDTD



Tabela 10 – Velocidades de transferência estimadas para o barramento local e para o rCUDA

<b>Barramento</b>	<b>Velocidade obtida</b>
PCI-e	6,44 MB/s
Gigabit Ethernet (TCP/IP)	0,15 MB/s

Na Equação de Laplace aumento do formato do stencil de 3 para 5 pontos implicou na ampliação do *speedup* obtido pelas versões CUDA, porém o mesmo não foi observado após o aumento para 9 pontos, indicando a necessidade de otimizações no *kernel* CUDA, de modo a promover o reuso das informações entre as *threads*.

Foi observada uma redução entre 54 e 58% do tempo de processamento ao dobrar a quantidade de placas por meio do rCUDA. Ao utilizar três GPUs foi possível obter valores de *speedup* superiores a 12 no experimento com a equação de Laplace e 79 na aplicação baseada no FDTD.

Outro fator importante observado foi que a utilização das múltiplas placas requereu, além do particionamento do domínio do problema, apenas a inserção de chamadas ao método *cuda-SetDevice()* e definição de *streams* paralelos para a execução simultânea entre as GPUs.

Os experimentos realizados com o rCUDA destacam ainda o impacto da velocidade da rede no desempenho geral da aplicação, sendo o rCUDA adequado em situações nas quais há uma baixa quantidade de dados sendo transferidos. O suporte ao padrão Infiniband® é implementado pelo rCUDA para prover acesso direto à memória via rede quando este tipo de infraestrutura estiver disponível (RCUDA, 2016).

Observa-se que a aplicação baseada no FDTD obteve resultados superiores de *speedup* em relação à implementação da Equação de Laplace ao utilizar 3 GPUs. Os resultados dos experimentos sugerem que tal diferença deve-se ao fato da aplicação FDTD inicializar a maior parte dos dados diretamente na memória da GPU, diminuindo o impacto da transferência de dados no desempenho geral da aplicação.

## 5 CONCLUSÕES E TRABALHOS FUTUROS

A partir dos estudos apresentados no presente trabalho é possível observar a importância da utilização de recursos complementares no desenvolvimento de aplicações baseadas em plataformas heterogêneas, especificamente no caso das GPGPUs, quando aplicadas à computação científica.

As ferramentas e técnicas apresentadas mostraram-se eficientes na diminuição da complexidade exposta ao programador de tais aplicações, permitindo a este obter proveito do aumento do desempenho proporcionado pelos Processadores Gráficos de forma mais transparente, sem que seja necessário aplicar-se profundamente a lidar com os detalhes de implementação e manipulação do hardware heterogêneo.

Os experimentos com o OpenACC mostraram a possibilidade de utilização de um modelo de programação mais simplificado, que apesar de resultar em um desempenho sub-ótimo em relação à uma aplicação escrita diretamente em CUDA, proporciona uma redução significativa no tempo de processamento, apresentando um desempenho superior inclusive ao obtido por meio de meio do método baseado em CPU com abordagem de programação semelhante (OpenMP).

Os resultados dos experimentos com o rCUDA indicam a possibilidade de utilização de MultiGPU em rede, a partir de equipamentos de menor custo, por meio de um recurso de *software* capaz de fornecer abstração à camada de rede, permitindo o desenvolvimento de aplicações que possam ser posteriormente executadas em equipamentos que possuam múltiplas placas GPU em de memória compartilhada, ou ainda permitindo a distribuição de problemas maiores por várias GPUs sem a necessidade de utilização de um sistema de passagem de mensagens, como o MPI.

### 5.1 Contribuições

Os estudos apresentados neste trabalho visam contribuir com a ampliação da utilização de aceleradores gráficos na computação científica ao demonstrar a viabilidade de obtenção de ganhos de desempenho mesmo nas fases iniciais da pesquisa das aplicações.

A pesquisa com o OpenACC explorou a possibilidade de utilização de novas técnicas de programação em GPGPU. O estudo apontou, por meio de uma avaliação de desempenho, a viabilidade de uma ferramenta potencialmente útil à computação científica.

A pesquisa com o software rCUDA produziu como contribuição a avaliação de uma proposta de arquitetura, baseada no software utilizado, capaz de oferecer recursos de MultiGPU a um baixo custo, por meio de uma rede local, e sem a necessidade de utilização de uma camada adicional de passagem de mensagens, ao utilizar de forma transparente os recursos de programação MultiGPU da linguagem CUDA.

Os casos abordados na presente pesquisa, portanto, resultam em contribuição para o desenvolvimento de técnicas aplicáveis a problemas reais na Pesquisa Científica, pois mesmo sendo

relativamente simples, as aplicações utilizadas neste estudo são baseadas em algoritmos utilizados em problemas reais do eletromagnetismo e permitem a observação das características exploradas através das técnicas apresentadas.

## **5.2 Trabalhos Futuros**

Como trabalho futuro, pretende-se aprimorar os métodos implementados, aplicando-os efetivamente ao contexto dos programas reais, proporcionando a redução do tempo de execução dos mesmos. Como alternativa para a melhoria dos resultados com o CUDA, pretende-se aplicar otimizações, por meio de outras estratégias de paralelização ou através do uso de bibliotecas específicas para álgebra linear, como por exemplo a CuBLAS (NVIDIA, 2013).

Como trabalho futuro relacionado ao rCUDA, pretende-se analisar o comportamento do desempenho ao usar um número maior de GPUs, a fim de investigar os limites de escalabilidade da técnica apresentada.

Um outro trabalho futuro possível consiste em efetuar uma comparação dos resultados obtidos no presente estudo com outras técnicas de computação multiGPU, como o uso de sistemas com múltiplas placas ou implementações utilizando MPI.

## REFERÊNCIAS

- ABDELKHALEK, R. et al. Fast seismic modeling and reverse time migration on a gpu cluster. In: HPCS, 2009. **Anais...** IEEE, 2009. p. 36–43.
- ARAUJO, J. d. S. **Desenvolvimento de metodologias para a localização de intruso em ambientes indoor**. 2010. Tese (Doutorado em Ciência da Computação) — Universidade Federal do Pará, Programa de Pós-Graduação em Engenharia Elétrica, 2010. Tese (Doutorado).
- BARBOSA, J. **Noções sobre matrizes e sistemas de equações lineares**. 2a edição. ed. [S.l.]: FEUP Edições, 2011.
- BÉNYÁSZ, G.; CSER, L. Clustering financial time series on cuda. In: CONFERENCE OF PHD STUDENTS IN COMPUTER SCIENCE INSTITUTE OF INFORMATICS OF THE UNIVERSITY OF SZEGED, 2010, Hungary. **Anais...** [S.l.: s.n.], 2010.
- CASANOVA, H.; LEGRAND, A.; ROBERT, Y. **Parallel algorithms**. [S.l.]: CRC Press, 2008. (Chapman & Hall/CRC Numerical Analysis and Scientific Computing Series).
- CECILIA, J. M.; GARCÍA, J. M.; UJALDÓN, M. Cuda 2d stencil computations for the jacobi method. In: INTERNATIONAL CONFERENCE ON APPLIED PARALLEL AND SCIENTIFIC COMPUTING - VOLUME PART I, 10., 2012, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2012. p. 173–183. (PARA'10).
- CHANDRA, R. et al. **Parallel programming in openmp**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.
- CHAPRA, S.; CANALE, R. **Métodos numéricos para engenharia - 7ª edição**. [S.l.]: McGraw Hill Brasil, 2016.
- CHENG, J.; GROSSMAN, M.; MCKERCHER, T. **Professional cuda c programming**. [S.l.]: Wiley, 2014. (EBL-Schweitzer).
- COOK, S. **Cuda programming: a developer's guide to parallel computing with gpus**. [S.l.]: Morgan Kaufmann, 2013. (Applications of GPU Computing Series).
- DIRK, S.; ATLE, V.; S., M. M. Evaluating openmp performance on thousands of cores on the numascale architecture. **Advances in Parallel Computing**, [S.l.], v. 27, n. Parallel Computing: On the Road to Exascale, p. 83–92, 2016.
- DONGARRA, J. et al. **High performance computing: technology, methods and applications: technology, methods and applications**. [S.l.]: Elsevier Science, 1995. (Advances in Parallel Computing).
- DUBOIS, M.; ANNAVARAM, M.; STENSTRÖM, P. **Parallel computer organization and design**. [S.l.]: Cambridge University Press, 2012. (Parallel Computer Organization and Design).
- ERLANGGA, Y. A. **A robust and efficient iterative method for the numerical solution of the helmholtz equation**. Delft University: fl.126, 2005. Tese (Doutorado).

- FARBER, R. **The openacc execution model**. Disponível em: <http://www.drdoobbs.com/parallel/the-openacc-execution-model/240006334>. Acesso em setembro de 2016.
- GEIST, A. **Pvm: parallel virtual machine :a users' guide and tutorial for networked parallel computing**. [S.l.]: MIT Press, 1994. (Scientific and engineering computation).
- GILAT, A.; SUBRAMANIAM, V. **Metodos numéricos para engenheiros e cientistas: uma introdução com aplicações usando o matlab**. [S.l.]: Bookman, 2008.
- GRIFFITHS, D. **Introduction to electrodynamics**. [S.l.]: Pearson, 2013. (Always learning).
- GROPP, W. et al. **Using advanced mpi: modern features of the message-passing interface**. [S.l.]: MIT Press, 2014. (Computer science & intelligent systems).
- HEROUX, M.; RAGHAVAN, P.; SIMON, H. **Parallel processing for scientific computing**. [S.l.]: Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 2006. (SIAM e-books).
- HISTENCILS. **Histencils**. Disponível em: <http://www.exastencils.org/histencils/2016/>. Acesso em setembro de 2016.
- HWU, W. **Heterogeneous system architecture: a new compute platform infrastructure**. [S.l.]: Elsevier Science, 2015.
- JAMSHED, S. **Using hpc for computational fluid dynamics: a guide to high performance computing for cfd engineers**. [S.l.]: Elsevier Science, 2015.
- KIRK, D. B.; HWU, W.-m. W. **Programming massively parallel processors: a hands-on approach**. 2st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.
- LENOSKI, D.; WEBER, W. **Scalable shared-memory multiprocessing**. [S.l.]: Elsevier Science, 2014.
- MANAVSKI, S.; VALLE, G. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. **BMC Bioinformatics**, [S.l.], v. 9, p. S10, 2008.
- MANFROI, L. L. F. et al. Avaliação de arquiteturas manycore e do uso da virtualização de gpus. In: XXXIV CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 2014. **Anais...** [S.l.: s.n.], 2014. p. 1837–1850.
- NVIDIA. **Cuda toolkit documentation**. Disponível em: <http://http://docs.nvidia.com/cuda/>. Acesso em novembro de 2016.
- OPENACC. **Openacc**. Disponível em: <http://www.openacc-standard.org/>. Acesso em novembro de 2016.
- OPENMP. **Openmp**. Disponível em: <http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>. Acesso em novembro de 2016.
- PACHECO, P. **Parallel programming with mpi**. [S.l.]: Morgan Kaufmann Publishers, 1997.

- PAN, L.; GU, L.; XU, J. Implementation of medical image segmentation in cuda. In: INFORMATION TECHNOLOGY AND APPLICATIONS IN BIOMEDICINE, 2008. ITAB 2008. INTERNATIONAL CONFERENCE ON, 2008. **Anais...** [S.l.: s.n.], 2008. p. 82–85.
- PITAKSIRIANAN, N.; NOURI, Z.; TU, Y. C. Efficient 2-body statistics computation on gpus: parallelization amp; beyond. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING (ICPP), 2016., 2016. **Anais...** [S.l.: s.n.], 2016. p. 380–385.
- RAHMAN, R. **Intel xeon phi coprocessor architecture and tools: the guide for application developers**. [S.l.]: Apress, 2013. (The expert's voice in microprocessors).
- RAUBER, T.; RÜNGER, G. **Parallel programming: for multicore and cluster systems**. [S.l.]: Springer Berlin Heidelberg, 2013.
- RCUDA, T. **rcuda**. Disponível em: <http://www.rcuda.net>. Acesso em novembro de 2016.
- REAÑO, C. et al. Improving the user experience of the rcuda remote gpu virtualization framework. **Concurrency and Computation: Practice and Experience**, [S.l.], v. 27, n. 14, p. 3746–3770, 2015.
- REYES, R. et al. accull: an openacc implementation with cuda and opencl support. In: EURO-PAR, 2012. **Anais...** Springer, 2012. p. 871–882. (Lecture Notes in Computer Science, v. 7484).
- SADIKU, M. **Numerical techniques in electromagnetics, second edition**. [S.l.]: Taylor & Francis, 2000.
- SALSA, S. **Partial differential equations in action: from modelling to theory**. [S.l.]: Springer International Publishing, 2015. (UNITEXT).
- SANDERS, J.; KANDROT, E. **Cuda by example: an introduction to general-purpose gpu programming**. 1st. ed. [S.l.]: Addison-Wesley Professional, 2010.
- SOUZA, D. L. et al. Pso-gpu: accelerating particle swarm optimization in cuda-based graphics processing units. In: GECCO (COMPANION), 2011. **Anais...** ACM, 2011. p. 837–838.
- SOUZA, D. L. et al. A novel competitive quantum-behaviour evolutionary multi-swarm optimizer algorithm based on CUDA architecture applied to constrained engineering design. In: SWARM INTELLIGENCE - 9TH INTERNATIONAL CONFERENCE, ANTS 2014, BRUSSELS, BELGIUM, SEPTEMBER 10-12, 2014. PROCEEDINGS, 2014. **Anais...** [S.l.: s.n.], 2014. p. 206–213.
- WANG, G. et al. **Algorithms and architectures for parallel processing: 15th international conference, ica3pp 2015, zhangjiajie, china, november 18-20, 2015, proceedings**. [S.l.]: Springer International Publishing, 2015. n. pt. 3. (Lecture Notes in Computer Science).
- WANG, Z. J. High-order computational fluid dynamics tools for aircraft design. **Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences**, [S.l.], v. 372, n. 2022, 2014.

- XIE, P.; WU, Y.; CHENG, K. The implementation of a network traffic detection model based on gpu and cpu heterogeneous platform. In: INTERNATIONAL CONFERENCE ON INFORMATION SCIENCES, MACHINERY, MATERIALS AND ENERGY (ICISMME 2015), 2015. **Anais...** [S.l.: s.n.], 2015.
- YANG, L.; GUO, M. **High-performance computing**: paradigm and infrastructure. [S.l.]: Wiley, 2005. (Wiley Series on Parallel and Distributed Computing).
- YEE, K. Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media. **IEEE Trans. Antennas and Propagation**, [S.l.], v. 14, p. 302–307, 1966.

## APÊNDICE A – MÉTODO DA ELIMINAÇÃO GAUSSIANA

A forma generalizada de um sistema linear é apresentada na Equação (A.1).

$$\begin{aligned}
 E_1 : a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1, \\
 E_2 : a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2, \\
 &\vdots \\
 E_n : a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n,
 \end{aligned} \tag{A.1}$$

Para resolver sistemas lineares, pode-se usar tanto métodos iterativos como métodos diretos (GILAT; SUBRAMANIAM, 2008). Métodos diretos são baseados na Eliminação Gaussiana (ERLANGGA, 2005) e calculam a solução exata de um sistema linear, (exceto por erros de arredondamento), com um número finito e bem determinado de operações aritméticas. São adequados para resolver sistemas lineares densos.

Métodos iterativos calculam a solução aproximada de um sistema linear e se beneficiam da esparsidade da matriz, exigindo pouca memória alocada, porém têm baixa velocidade de convergência para encontrar a solução do sistema linear, dependendo do condicionamento da matriz, e em alguns casos podem divergir. Desse modo, a solução por métodos baseados na Eliminação Gaussiana, apesar de demandar maior esforço computacional, é mais adequada nos casos em que as características do problema permitirem sua aplicação (CHAPRA; CANALE, 2016). A paralelização do método de Eliminação Gaussiana permite a diminuição do seu tempo de execução, tornando viável o seu uso na obtenção da solução exata de sistemas de grande porte, como os utilizados no processamento sísmico e outras áreas da computação científica.

### A.0.1 Descrição do Método

A Eliminação Gaussiana consiste na realização de transformações em um sistema linear, de modo a produzir um sistema equivalente, cuja solução possa ser obtida por substituição reversa. Seja um sistema linear da forma  $Ax = b$ , onde cada elemento  $a_{i,j}^k$  pertença a matriz  $A$  de coeficientes, e cada elemento  $b_i^k$  pertença ao vetor de termos independentes  $b$ , as Equações (A.2) e (A.3), definem as  $k$  operações de eliminação necessárias à obtenção de cada um dos elementos  $a_{i,j}^k$  e  $b_i^k$ , respectivamente:

$$a_{i,j}^k = a_{i,j}^{k-1} - \frac{a_{i,k-1}^{k-1}}{a_{k-1,k-1}^{k-1}} a_{k-1,j}^{k-1}, k = 2, \dots, n \tag{A.2}$$

$$b_i^k = b_i^{k-1} - \frac{a_{i,k-1}^{k-1}}{a_{k-1,k-1}^{k-1}} b_{k-1}^{k-1}, k = 2, \dots, n \tag{A.3}$$



## APÊNDICE B – MÉTODO DA EQUAÇÃO DE LAPLACE

A abordagem do método permite a representação algébrica do problema, de modo que o valor da solução em um determinado ponto possa ser obtido de forma aproximada, baseado nos valores de pontos vizinhos.

De acordo com SADIKU (2000), a solução de um problema por meio do MDF envolve basicamente três passos:

1. Dividir a região de solução em uma grade de nós
2. Aproximar a equação diferencial dada por seu equivalente em diferenças finitas que relaciona a variável dependente em um ponto na região da solução com seus valores nos pontos vizinhos
3. Resolver as equações de diferenças, sujeitas às condições de contorno prescritas e/ou condições iniciais

Com o objetivo de ilustrar a aplicação do método, tomemos como exemplo uma função  $f(x)$ , representada pela Figura 25.

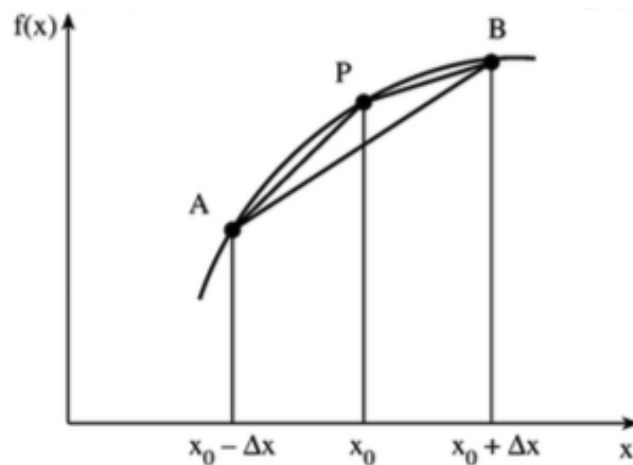


Figura 25 – Aproximações para a derivada de  $f(x)$  (Adaptado de (SADIKU, 2000))

Podemos aproximar a derivada no ponto P pela inclinação do arco PB (diferença avançada).

$$f'(x_0) \simeq \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x} \quad (\text{B.1})$$

Ou pela inclinação do arco AP (diferença atrasada).

$$f'(x_0) \simeq \frac{f(x_0) - f(x_0 - \Delta x)}{\Delta x} \quad (\text{B.2})$$

Ou ainda pela inclinação do arco AB (diferença centrada).

$$f'(x_0) \simeq \frac{f(x_0 + \Delta x) - f(x_0 - \Delta x)}{2\Delta x} \quad (\text{B.3})$$

Desse modo, é possível aproximar a segunda derivada de  $f(x)$  por meio de

$$f''(x_0) \simeq \frac{f'(x_0 + \Delta x/2) - f'(x_0 - \Delta x/2)}{\Delta x} \quad (\text{B.4})$$

Obtendo-se

$$f''(x_0) \simeq \frac{f(x_0 + \Delta x) - 2f(x_0) + f(x_0 - \Delta x)}{(\Delta x)^2} \quad (\text{B.5})$$

E analogamente, para a dimensão  $y$  temos

$$f''(y_0) \simeq \frac{f(y_0 + \Delta y) - 2f(y_0) + f(y_0 - \Delta y)}{(\Delta y)^2} \quad (\text{B.6})$$

**ANEXO A – ARTIGO PUBLICADO**

GONÇALVES, Nielsen A. et al. **Comparação e Análise de Desempenho de Aceleradores Gráficos no Processamento de Matrizes**. XXXV CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO - XIV WPERFORMANCE, 2015