UNIVERSIDADE DE BRASÍLIA
FACULDADE GAMA / FACULDADE DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INTEGRIDADE DE MATERIAIS DA ENGENHARIA

SOLUÇÃO NUMÉRICA MASSIVAMENTE PARALELA DE PROBLEMAS POTENCIAIS UTILIZANDO O MÉTODO DOS ELEMENTOS DE CONTORNO

ALISON BARROS DA SILVA

ORIENTADOR(A): DR. RAFAEL MORGADO SILVA
CO-ORIENTADOR(A): DR. MANUEL NASCIMENTO DIAS BARCELOS JÚNIOR

DISSERTAÇÃO DE MESTRADO EM INTEGRIDADE DE MATERIAIS DA ENGENHARIA

PUBLICAÇÃO: 003A/2013
BRASÍLIA/DF: 04 – 2013

UNIVERSIDADE DE BRASÍLIA
FACULDADE GAMA / FACULDADE DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INTEGRIDADE DE MATERIAIS DA ENGENHARIA.

ALISON BARROS DA SILVA

SOLUÇÃO NUMÉRICA MASSIVAMENTE PARALELA DE PROBLEMAS POTENCIAIS UTILIZANDO O MÉTODO DOS ELEMENTOS DE CONTORNO

DISSERTAÇÃO DE MESTRADO SUBMETIDA AO PROGRAMA DE PÓS-GRADUAÇÃO EM INTEGRIDADE DE MATERIAIS DA ENGENHARIA DA FACULDADE GAMA E FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA, COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM INTEGRIDADE DE MATERIAIS DA ENGENHARIA.

ORIENTADOR: DR. RAFAEL MORGADO SILVA
CO-ORIENTADOR: DR. MANUEL NASCIMENTO DIAS BARCELOS JÚNIOR

BRASÍLIA
2013

UNIVERSIDADE DE BRASÍLIA
FACULDADE GAMA / FACULDADE DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INTEGRIDADE DE MATERIAIS DA ENGENHARIA.

SOLUÇÃO NUMÉRICA MASSIVAMENTE PARALELA DE PROBLEMAS POTENCIAIS UTILIZANDO O MÉTODO DOS ELEMENTOS DE CONTORNO

ALISON BARROS DA SILVA

DISSERTAÇÃO DE MESTRADO SUBMETIDA AO PROGRAMA DE PÓS-GRADUAÇÃO EM INTEGRIDADE DE MATERIAIS DA ENGENHARIA DA FACULDADE GAMA E FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA, COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM INTEGRIDADE DE MATERIAIS DA ENGENHARIA.

APROVADA POR:

_____
Prof. Dr. Rafael Morgado Silva
(Orientador)

_____
Prof. Dr. Eder Lima de Albuquerque
(Examinador Interno)

_____
Prof. Dr. João Luiz Azevedo de Carvalho
(Examinador Externo)

BRASÍLIA, 05 DE ABRIL DE 2013.

BRASÍLIA/DF, 05 DE ABRIL DE 2013.

FICHA CATALOGRÁFICA

ALISON BARROS DA SILVA
SOLUÇÃO NUMÉRICA MASSIVAMENTE PARALELA DE PROBLEMAS POTENCIAIS UTILIZANDO O MÉTODO DOS ELEMENTOS DE CONTORNO, [Distrito Federal] 2013.
003A/2013, 210 x 297 mm (FGA/UnB Gama, Mestre, Integridade de Materiais da Engenharia, 2013). Dissertação de Mestrado - Universidade de Brasília. Faculdade Gama. Programa de Pós-Graduação em Integridade de Materiais da Engenharia.

1. EQUAÇÃO DE LAPLACE	2. MÉTODO DOS ELEMENTOS DE CONTORNO
3. JACOBI	4. GPU
I. FGA UnB Gama/ UnB.	II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

BARROS, A. (2013). SOLUÇÃO NUMÉRICA MASSIVAMENTE PARALELA DE PROBLEMAS POTENCIAIS UTILIZANDO O MÉTODO DOS ELEMENTOS DE CONTORNO. Dissertação de Mestrado em Integridade de Materiais da Engenharia, Publicação 003A/2013, Programa de Pós-Graduação em Integridade de Materiais da Engenharia, Faculdade Gama, Universidade de Brasília, Brasília, DF, p. 86.

CESSÃO DE DIREITOS

AUTOR: ALISON BARROS DA SILVA.
TÍTULO: SOLUÇÃO NUMÉRICA MASSIVAMENTE PARALELA DE PROBLEMAS POTENCIAIS UTILIZANDO O MÉTODO DOS ELEMENTOS DE CONTORNO.

GRAU: Mestre

ANO: 2013

2013
FACULDADE UNB/GAMA - ÁREA ESPECIAL N. 01, SETOR CENTRAL.
CEP 72.405-610 Brasília, DF – Brasil.

AGRADECIMENTOS

A Deus por ter me dado o sopro da vida. Agradeço pela sua presença constante em todas as áreas da minha vida, por me auxiliar nos momentos difíceis e se alegrar comigo nos momentos felizes, fazendo com que eu sempre sinta suas mãos agindo e fazendo com que eu me sinta amado e cuidado.

Aos meus pais Amaury Amaral da Silva e Clotilde Cruz Barros que sempre estiveram ao meu lado motivando, aconselhando e fornecendo todo o suporte, material e emocional, que um filho poderia ter desde a infância até os dias de hoje. A eles sou muito mais do que grato, devo tudo que tenho e o que sou.

Ao meu irmão Anderson Barros da Silva que sempre me incentivou nas minhas idéias e sempre me apoiou mostrando como tirar algo positivo das coisas, mesmo das piores situações que vivi durante o meu amadurecimento.

A minha loirinha Paulinha Oliveira que esteve presente desde o início da realização dessa dissertação. Obrigado por ter abdicado, juntamente comigo, de diversos finais de semana, feriados e festas. Nunca esquecerei os momentos em que as coisas ficaram muito difíceis e você conseguiu segurar o meu estresse, procurou me acalmar e sempre esteve disposta a ajudar no que fosse necessário, sempre acreditando que eu obteria sucesso. Te amo!

Ao meu grande amigo e orientador Dr. Rafael Morgado Silva que esteve sempre auxiliando, incentivando e, quando necessário, sendo duro com as palavras. Muito obrigado pelas madrugadas que passou comigo discutindo o trabalho, apresentando idéias novas e sempre motivando e acreditando em mim, até quando nem eu mesmo acreditava. Todo o acompanhamento durante a realização dessa dissertação me fez crescer muito como pessoa, como estudante e como profissional e você foi peça fundamental nesse crescimento. Muito obrigado!

Ao meu coorientador Dr. Manuel Nascimento Dias Barcelos Júnior por estar sempre disposto a auxiliar no que estivesse ao seu alcance, pelas reuniões à altas horas da noite e pelo acompanhamento nas madrugadas em diversas situações difíceis propondo soluções e discutindo os resultados obtidos durante a realização dessa dissertação, além de ter o pulso

firme quando foi necessário. Agradeço por ter feito parte desse crescimento juntamente com o Dr. Rafael Morgado Silva.

Ao Msc. Hilmer Rodrigues Neri, coordenador do projeto Desafio Positivo, que ajudou nos momentos corridos controlando as demandas a mim passadas permitindo que eu tivesse um pouco mais de tempo para a realização dessa dissertação. Gestos como esse jamais serão esquecidos.

Um agradecimento especial a Frederico Cabral de Menezes, meu chefe durante a realização dessa dissertação. Sem sua compreensão e apoio flexibilizando o horário de trabalho não haveria a possibilidade nem ao menos de começar a realização deste sonho. Eu não tenho palavras para descrever o quanto sou grato, de qualquer maneira, muito obrigado!

Aos meus colegas de trabalho Evandro Alves Rodrigues, Aderbal Botelho Neto, Thiago Montenegro, Werberth Silva e Erik Galletti. Em diversos momentos durante a realização deste trabalho, foram vocês que tomaram para si responsabilidades que eram minhas para que eu tivesse mais tempo. Sou muito grato a vocês.

Aos meus grandes e preciosos amigos que por diversas vezes, a vida nos manteve distantes tornando-os praticamente amigos invisíveis, como os defino gentilmente. O apoio, discussões, lamentações e bebedices juntos foram essenciais durante essa fase da minha vida, e fico muito feliz de tê-los como amigos para compartilhar todos os momentos com vocês, Fernando Barbosa e Ulisses Ziech.

Ao grupo do Laboratório de Supercomputação para Sistemas Complexos, coordenado pelo Dr. Fernando Albuquerque de Oliveira, pelo incentivo e longas conversas desde o momento que nos conhecemos. O ambiente proporcionado pelo grupo me fez enxergar o quão gratificante é a carreira científica.

Por último, agradeço a todos aqueles que de alguma forma contribuíram para o êxito desse trabalho.

RESUMO

Um problema potencial pode ser caracterizado como um problema da natureza cuja solução pode ser obtida através da Equação de Laplace, que é uma equação diferencial parcial de segunda ordem. A presença de problemas potenciais na natureza fez com que fosse desenvolvida uma área de pesquisa dedicada ao seu estudo. Em problemas de múltiplas dimensões o tratamento analítico pode ser inviável, sendo assim, é comum o uso de modelagem numérica a fim de obter suas soluções.

Existem diversos modelos numéricos capazes de resolver a Equação de Laplace, dentre eles estão o Método dos Elementos Finitos (MEF), Método dos Volumes Finitos (MVF), Método das Diferenças Finitas (MDF) e Método dos Elementos de Contorno (MEC). Dentre os métodos citados, o MEC é o mais recente, e atualmente está sendo muito utilizado na resolução de problemas de grandes dimensões e de domínios semi-infinitos.

O MEC utiliza uma formulação matemática baseada no Teorema de Green a fim de reduzir uma dimensão do problema. Assim, é possível obter um ganho no custo computacional, apesar do esforço matemático ser maior. Apesar do ganho obtido, as matrizes geradas pelo método são cheias e não-simétricas, fazendo com que o custo computacional ainda seja elevado.

Devido à crescente exigência dos usuários de computadores no quesito qualidade gráfica, as fabricantes desse tipo de hardware se viram forçadas a desenvolver novas tecnologias capazes de suprir essa demanda, surgindo assim, as placas gráficas com processadores e memórias dedicadas. Este tipo de hardware chamou a atenção da comunidade científica por ser paralelo por natureza, sendo capaz de obter um desempenho comparado a um supercomputador.

O presente trabalho visa a implementação de uma biblioteca paralela adaptada a estrutura de placas gráficas para resolução de sistemas de equações lineares obtidos a partir da discretização de problemas potenciais com o MEC, utilizando a linguagem de programação OpenCL, a fim de avaliar a viabilidade de seu uso em ambiente híbrido, ou seja, contendo uma ou mais *Central Processing Units* (CPUs) e *Graphics Processing Units* (GPUs). A implementação foi validada, sendo aplicada ao problema de um fluido

potencial em torno de um cilindro circular impenetrável e diversas técnicas de otimização do algoritmo foram avaliadas de forma a fornecer uma base de conhecimento para futuros trabalhos que venham utilizar GPUs.

Os resultados mostram que uma implementação do método iterativo de Jacobi, que é utilizado na resolução de sistemas lineares, com paralelização trivial, semelhante ao problema de N-Corpos (N-Body), não oferece um desempenho expressivo que justifique o uso de computação massivamente paralela, por outro lado, utilizando as técnicas de otimizações apresentadas, é possível obter um ganho de até 5.5 vezes em relação ao algoritmo serial. Além disso, o trabalho aponta limitações na alocação de memória disponibilizada pela implementação OpenCL da fabricante AMD ATI.

Palavras-chave: Equação de Laplace; Método dos elementos de contorno; Jacobi; Computação de alto desempenho; Programação massivamente paralela; GPU; OpenCL;

ABSTRACT

A potential problem could be defined as a nature problem that satisfies The Laplace's Equation, that is, a second order differential equation. The significant presence of potential problems in nature made it necessary to develop a research area dedicated to its study. This kind of problem can be solved using Laplace's Equation as a tool, a partial second order differential equation. Analytical approaches may be impractical for higher dimensions, so it is common to use numerical modeling to obtain solutions.

There are many numerical models capable of solving Laplace's Equations, among them the Finite element method (FEM), Finite Volume Method (FVM), Finite Difference method (FDM) and the Boundary Element Method (BEM). Among these, the BEM is the most recent and is currently being used for higher dimensions and semi-infinite domain problems.

The BEM uses a mathematical formulation based on Green's Theorem to reduce one dimension of the problem, and as such, makes it possible to obtain computational gain, in spite of the higher mathematical efforts. A side effect of this gain is that the generated matrices are full and asymmetrical, making the computational cost to be still high.

Computer user's demand for higher quality graphics push manufactures to develop new technologies, suck as discrete graphic boards with dedicates processors an memory. This kind of hardware drew attention from the scientific community for its parallel nature being capable of performing Teraflop order of magnitude calculations for single precision math and Gigaflops order of magnitude calculations for double precision.

This works intends to implement a parallel library using the BEM on graphic boards using the OpenCL programming language, to evaluate the viability of a hybrid (CPU and GPU) environment. The implementation was validated by being applied to a potential fluid around a solid cylinder and many algorithm optimization techniques was implemented to create a knowledge base to futures works using GPUs.

The results show that an implementation of Jacobi's iterative method, normally used on linear system solving, and a trivial parallelization similar to the one used on the N-Body solution does not show an expressive performance to justify massive parallel computing,

due to the high number of memory accesses. Although, using the suggested optimization techniques, it is possible to reach a 5.5 gain when compared the same algorithm running serially.

Keywords: Laplace's Equation, Boundary Element Method, Jacobi, High performance Computing, Massive parallel programing, GPU, OpenCL;

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

API	Application Programming Interface
BEM	Boundary Elements Method
BIE	Boundary Integral Equation
CFD	Computational Fluid Dynamics
CPU	Central Processing Unit
FEM	Finite Element Method
FVM	Finite Volume Method
GB	Gigabyte
$GBPS$	Gigabit por segundo
$GDDR$	Graphics Double Data Rate
GPU	Graphics Processing Unit
KB	Kilobytes
$KBPS$	Kilobits por segundo
RAM	Random Access Memory
$SIMD$	Single Instruction Multiple Data
$SISD$	Single Instruction Single Data
MDF	Método das Diferenças Finitas
MEC	Método dos Elementos de Contorno
MEF	Método dos Elementos Finitos
$MISD$	Multiple Instruction Single Data

$MIMD$	Multipe Instruction Multiple Data
MPI	Message Passing Interface
MVF	Método dos Volumes Finitos
TPU	Quantidade de stream cores
ULA	Unidade Lógico Aritmética

Sumário

1 INTRODUÇÃO

Problemas potenciais são definidos como problemas cuja solução pode ser obtida através da Equação de Laplace. A grande variedade deste tipo de problema na natureza fez com que surgisse a chamada teoria de potenciais, onde são estudadas as funções que fornecem a solução para os problemas potenciais, juntamente com suas características.

A Equação de Laplace é uma equação diferencial parcial de segunda ordem linear. Recebe esse nome devido ao primeiro estudioso de suas propriedades chamado Pierre-Simon Laplace. Uma função que satisfaz a Equação de Laplace é chamada de função harmônica.

Em coordenadas cartesianas e em duas dimensões a Equação de Laplace geralmente é escrita da seguinte forma:

$$\nabla^2 \phi = 0, \tag{1.1}$$

onde $\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$ é o operador Laplaciano.

Como dito anteriormente, a Equação de Laplace possui aplicação nas mais variadas áreas da ciência, como no projeto de dispositivos microeletrônicos para detecção de semicondutores (1), na distribuição de temperatura em um dissipador semi-infinito (2), ou na análise de campos de fluxos em mecânica dos fluidos (3), dentre outras aplicações. A fim de atingir o objetivo deste trabalho e desenvolver o modelo numérico, utilizando o MEC, para a resolução de problemas potenciais, será utilizado, como base, a aplicação da Equação de Laplace bidimensonal na resolução do problema de um fluido potencial.

Utilizando como exemplo um fluido potencial, ou seja, invíscido e irrotacional, pode-se estabelecer as relações analíticas que descrevem o movimento do fluido. Supondo que o fluido esteja a uma velocidade de referência $\vec{V} = v_x\hat{i} + v_y\hat{j} = v_x\hat{i}$, e escoando em torno de um cilindro estático, é possível obter o campo de velocidades próximo ao cilindro e a pressão imposta pelo fluido, fazendo as seguintes considerações:

$$\vec{V} \cdot \hat{n} = 0,$$

$$\nabla \times \vec{V} = 0.$$

A primeira equação se refere a condição de não ser permitido que uma linha de corrente faça interseção com outra linha de corrente. A segunda se refere ao fato do escoamento ser irrotacional.

Utilizando o fato do escoamento ser irrotacional, pela física matemática (4), é possível dizer que existe uma função potencial ϕ tal que:

$$\vec{V} = \nabla\phi. \tag{1.2}$$

Como o fluido é incompressível ($\nabla\cdot\vec{V} = 0$), então a velocidade potencial deve satisfazer a Equação de Laplace $\nabla^2\phi = 0$. Assim, para o caso bidimensional em coordenadas polares com simetria radial, tem-se:

$$\frac{\partial^2\phi}{\partial r^2} + \frac{1}{r}\frac{\partial\phi}{\partial r} + \frac{1}{r^2}\frac{\partial^2\phi}{\partial\theta^2} = 0. \tag{1.3}$$

Para as condições de contorno impostas, a função velocidade potencial é escrita como:

$$\phi(r,\theta) = U\left(\frac{r+R^2}{r}\right)\cos(\theta), \tag{1.4}$$

onde U é a velocidade de corrente livre e R é o raio do cilindro.

Pode-se, então, extrair as informações de velocidade radial e angular através das seguintes relações:

$$V_r = \frac{\partial\phi}{\partial r} = U\left(1-\frac{R^2}{r^2}\right)\cos(\theta),$$
$$V_\theta = \frac{1}{r}\frac{\partial\phi}{\partial\theta} = -U\left(1+\frac{R^2}{r^2}\right)\text{sen}(\theta).$$

A velocidade total é dada por $\vec{V} = \vec{V_r} + \vec{V_\theta}$, que é uma soma vetorial.

De posse desses resultados, pode-se utilizar a equação de Bernoulli que relaciona a velocidade e pressão de fluidos para obter a pressão em qualquer ponto do escoamento da seguinte maneira:

$$p = \frac{1}{2}\rho\left(U^2 - V^2\right) + p_\infty, \tag{1.5}$$

onde p_∞ é a pressão de referência, ρ a densidade específica, V é a velocidade do fluido no

ponto considerado e U é a velocidade de referência. Substituindo $\vec{V} = V_r + V_\theta$ e fazendo algumas manipulações algébricas, chega-se ao resultado:

$$p = \frac{1}{2}\rho U^2 \left(\frac{R^2}{r^2}\cos(2\theta) - \frac{R^4}{r^4} \right) + p_\infty, \tag{1.6}$$

que pode ser utilizado para representar o problema graficamente, obtendo uma forma semelhante à da figura 1.

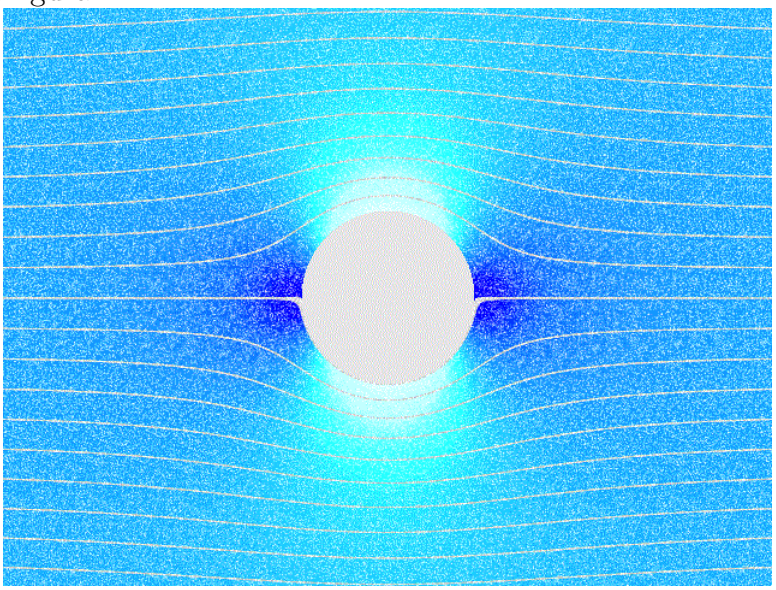


Figura 1: Exemplo de escoamento de um fluido potencial.

Observa-se na, figura 1, uma pressão maior no bordo de ataque, ou seja, na região frontal da esquerda para a direita, enquanto que nos contornos superior e inferior do cilindro existem regiões de baixas pressões.

Estes resultados serão usados como parâmetros para validar os resultados numéricos que serão obtidos nos próximos capítulos.

Com o desenvolvimento das poderosas tecnologias de computação, foram criados diversos algoritmos numéricos a fim de auxiliar a comunidade científica, abrindo assim um novo ramo de pesquisa chamado computação científica. Dentro dessa grande área de pesquisa, existe uma subárea que se concentra no estudo de simulações numéricas com aplicações em dinâmica dos fluidos, frequentemente chamada de dinâmica dos fluidos computacional (ou do inglês *Computational Fluid Dynamics* - CFD).

Diversos métodos numéricos foram desenvolvidos para auxiliar a dinâmica dos fluidos com uso de computadores. Dentre esses diversos métodos, podem ser destacados o método dos elementos finitos (MEF) (ou do Inglês, *Finite Element Method* - FEM), método dos volumes finitos (MVF) (ou do Inglês, *Finite Volume Method* - FVM), método das

diferenças finitas (MDF) (ou do Inglês, *Finite Difference Method* - FDM) e o método dos elementos de contorno (MEC) (ou do Inglês, *Boundary Element Method* - BEM). Esses métodos são constantemente utilizados para a resolução da equação de Navier–Stokes a fim de simular problemas de dinâmica dos fluidos.

O FEM é um método de resolução de equações diferenciais que consiste em dividir o domínio do problema em elementos contínuos, formando uma malha com as mesmas propriedades do domínio original. Assim, cada ponto da malha determina um nó onde será resolvida uma equação diferencial. Esse modelo é muito útil na resolução de problemas grandes. Porém, o custo computacional é muito alto, devido ao método de discretização da malha. Além disso, tratar problemas que envolvem meio infinito pode ser impraticável, devido à necessidade de discretização de todo o espaço a ser analisado (5). Utilizando esse método, é possível, por exemplo, estudar a interação fluido–estrutura em sistemas aerodinâmicos sujeitos a fluidos que apresentam turbulência, como descrito na referência (6).

O FVM utiliza os princípios de conservação de grandezas, durante um determinado processo, e os conceitos de volumes de controle discretos, ou células, para discretizar o domínio em malhas. Em seguida, é realizada a integração sob todos os volumes de controle desse domínio discretizado. Assim, a soma de todos os volumes discretos é igual ao volume do domínio (7). Devido à generalidade do método, ele pode ser aplicado a qualquer tipo de malha, estruturada ou não. Através do método de volumes finitos, utilizando a modelagem através da equação de conservação do momento, é possível, por exemplo, o estudo de sólidos incompressíveis ou quase incompressíveis, como mostra a referência (8). No referido trabalho, foi possível predizer um comportamento elástico linear de um sólido elástico incompressível.

Por outro lado, o MEC realiza uma discretização apenas no contorno do domínio, fazendo com que problemas com meio infinito e problemas com muitos graus de liberdade sejam tratados de maneira mais eficiente e com menor custo computacional, em relação aos outros métodos, uma vez que, sua formulação matemática diminui em um grau a ordem do problema. Porém, as matrizes resultantes deste método são cheias e não simétricas, fazendo com que, mesmo a otimização do método imposta pelo MEC seja custosa em nível de computação. Portanto, outras formas de otimização fazem-se necessárias, como foi realizado na referência (9), onde é proposto uma estratégia de paralelização a fim de melhorar o tempo de convergência de métodos iterativos aplicados ao método dos elementos de contorno.

A fim de otimizar os cálculos científicos, foram desevolvidos os *clusters* de computadores, que são agregados de computadores que fazem uso de seus recursos de forma colaborativa, atingindo níveis de processamento que somente um supercomputador de alto custo seria capaz de atingir. A utilização de *clusters* tornou-se uma tendência, prova disso é a quantidades de *clusters* que estão identificados na lista top500, que elege os computadores de mais alto desempenho do mundo, sendo atualizada semestralmente (10).

Atualmente, a comunidade científica têm mostrado grande interesse no uso de placas gráficas para otimizar os algoritmos numéricos nas mais diversas áreas. Essa tendência pode ser observada através da crescente quantidade de publicações de resultados de desempenhos obtidos. A título de exemplos, seguem algumas publicações neste sentido.

Na referência (11), foi desenvolvido um algoritmo massivamente paralelo para resolução de um problema de grande interesse em sistemas que apresentam difusão, o movimento browniano correlacionado. O ganho obtido neste trabalho foi de 93 vezes em relação ao mesmo algoritmo otimizado para unidade central de processamento (UCP) (ou do inglês *Central Processing Unit* (CPU))

Utilizando o conceito de múltiplas unidades gráficas de processamento (multiGPU), foi realizada, na referência (12), uma otimização do algoritmo de Monte Carlo–Metrópolis, a fim de obter a temperatura crítica do modelo de Ising em duas dimensões. Os resultados mostram um ganho de desempenho de 15 a 35 vezes se comparado ao mesmo algoritmo otimizado para CPU.

Na referência (13), foi apresentado o uso de unidade gráfica de processamento (UGP) (ou do inglês *Graphics Processing Unit* (GPU)) aplicado à engenharia ambiental. Através de um modelo Lagrangeano de partícula, foi simulado o transporte e transformação de um radionuclídeo — átomo com núcleo instável — de uma única fonte. Os resultados mostram um ganho de desempenho de 80 a 120 vezes quando comparado com o algoritmo serial executado em uma única CPU.

Como um último exemplo, tem-se a aplicação de GPU em dinâmica molecular. Paralelizando os cálculos de força de Lennard—Jones em um sistema de partículas, a referência (14) mostra um ganho de desempenho de 30 vezes comparado ao mesmo algoritmo executado em uma única CPU.

Em contextos como os citados acima, utilizando a atual tecnologia de computação e os modelos matemáticos desenvolvidos, é possível obter simulações extremamente realistas, através de linguagens de baixa curva de aprendizado, como apresenta a referência (15).

1.1 OBJETIVO DO TRABALHO

O objetivo deste trabalho é implementar uma biblioteca para a resolução da Equação de Laplace em duas dimensões através do método dos elementos de contorno, utilizando a linguagem paralela de uso geral OpenCL, tanto para CPU quanto para GPU, podendo ser executada em sistemas híbridos (CPU e GPU).

1.2 DIVISÃO DA DISSERTAÇÃO

- Capítulo 2: Aborda o método dos elementos de contorno, que é a principal ferramenta matemática utilizada neste trabalho. Após uma breve descrição das características do método, é feita a dedução da solução fundamental da Equação de Laplace, que é a equação que rege o escoamento de fluidos potenciais. Neste ponto, é deduzida a equação integral de contorno, que é a principal equação do método, cuja discretização leva ao modelo numérico utilizado. Também é apresentada a relação da equação com a localização dos pontos, sendo dado um tratamento diferenciado quando o ponto está localizado no contorno ou dentro dele.

- Capítulo 3: Apresenta os principais conceitos da programação massivamente paralela em placas de vídeo, com ênfase na linguagem de programação OpenCL e arquitetura de GPU Cypress.

- Capítulo 4: Apresenta a modelagem do algoritmo serial, além da modelagem do algoritmo paralelo com uma GPU e múltiplas GPUs utilizando a linguagem OpenCL, para o escoamento de um fluido potencial. Um fluxograma é apresentado explicando as funcionalidades dos diversos módulos do algoritmo e as otimizações realizadas. Além disso, são desenvolvidas todas as técnicas de otimização dos algoritmos em GPUs utilizadas.

- Capítulo 5: Apresentada os resultados do trabalho, utilizando gráficos para definição dos melhores parâmetros de grid do OpenCL, comparação de desempenho dos métodos de otimização implementados executados em CPU, com 1 e 8 núcleos, e GPU.

- Capítulo 6: Apresentada as considerações finais do trabalho, incluindo possíveis melhorias para trabalhos futuros.

2 MÉTODO DOS ELEMENTOS DE CONTONO

O MEC é um método que utiliza uma equação integral de contorno (ou do Inglês, *Boundary Integral Equation* - BIE) para resolver problemas de valores iniciais e de contorno. O objetivo do método é reduzir, em uma ordem, os graus de liberdade do problema, de modo que um problema com três dimensões possa ser tratado por equações em duas dimensões, ao passo que um problema em duas dimensões possa ser tratado com equações unidimensionais.

Apesar do método reduzir os graus de liberdade do problema, sua formulação apresenta matrizes cheias e não-simétricas, fazendo com que, mesmo sendo vantajoso em relação aos outros métodos — como o método dos elementos finitos ou método dos volumes finitos — possua um alto custo computacional. Apesar do resultado final ser mais eficiente, quando comparado aos métodos citados anteriormente, o custo de modelagem é alto, pois as formulações matemáticas são mais complexas e exigem o conhecimento da solução fundamental do problema.

Liu (16) apresenta algumas vantagens no uso do método de elementos de contorno, como:

- Acurácia: É um método muito preciso, devido ao seu desenvolvimento semianalítico;
- Modelagem eficiente: A malha do método é facilmente gerada em problemas de três dimensões ou problemas de domínio infinito, devido à redução dos graus de liberdade do problema; e
- Flexibilidade: O método pode ser acoplado a outro método, trazendo flexibilidade na resolução de problemas mais complexos.

Nesta seção, será apresentado o modelo clássico do método dos elementos de contorno.

2.1 SOLUÇÃO FUNDAMENTAL DA EQUAÇÃO DE LAPLACE

O MEC se baseia no conhecimento da solução fundamental do problema a ser tratado. Portanto, nesta seção será desenvolvida a solução fundamental da Equação de Laplace, objeto de estudo deste trabalho.

Suponha uma fonte localizada em $P(x, y)$ no plano x-y, cuja densidade $Q(\xi, \eta)$ possa ser representada pela função densidade de probabilidade δ como $f(Q) = \delta(Q - P)$, e que o potencial $v = v(Q, P)$ produzido satisfaça a seguinte equação:

$$\nabla^2 v = \delta(Q - P), \tag{2.1}$$

onde $v = v(Q, P)$, δ é a função delta de Dirac que será explicada adiante e x, y, ξ, η são as coordenadas globais dos pontos P e Q. A solução particular da equação 2.1 é chamada de solução fundamental da equação potencial.

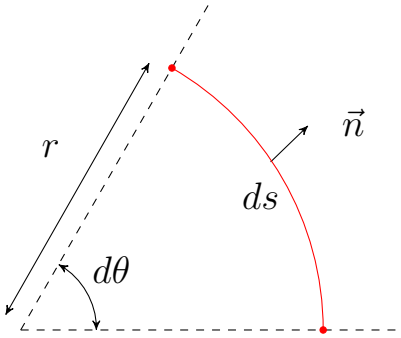
O problema analisado possui simetria radial, então é mais conveniente tratá-lo em coordenadas polares. Assim, sabendo que a função delta é nula para todos os pontos do domínio exceto na fonte $Q(\xi, \eta)$, a Equação de Laplace pode ser representada por:

$$\frac{1}{r}\frac{d}{dr}\left(r\frac{dv}{dr}\right) = 0. \tag{2.2}$$

Nesta equação, tem-se que $r = |Q - P| = \sqrt{(\xi - x)^2 + (\eta - y)^2}$. Integrando a equação 2.2 duas vezes obtém-se o resultado:

$$v = A \cdot \ln(r) + B, \tag{2.3}$$

onde A e B são constantes definidas pelas condições de contorno. A fim de obter a solução particular, pode-se considerar $B = 0$ enquanto que a constante A é obtida utilizando a simetria radial do problema da seguinte maneira:



$$\frac{\partial v}{\partial n} = \frac{\partial v}{\partial r} = \frac{A}{r},$$

$$ds = r \cdot d\theta.$$

Figura 2: Relações geométricas do elemento diferencial de contorno

onde ds é um elemento diferencial do contorno, $d\theta$ é o ângulo formado pelos pontos extremos do elemento diferencial de contorno e a origem e $\vec{n}$ é o vetor normal ao contorno. Essas relações serão utilizadas em seguida para obter a constante A.

A segunda identidade de Green afirma que:

$$\int_\Omega \left(v\nabla^2 u - u\nabla^2 v\right) d\Omega = \int_\Gamma \left(v\frac{\partial u}{\partial n} - u\frac{\partial v}{\partial n}\right) ds. \tag{2.4}$$

Utilizando as condições de contorno: $u = 1$ e $v = A \cdot \ln(r)$ e aplicando ao problema,

$$-\int_\Omega \nabla^2 v d\Omega = -\int_\Gamma \frac{\partial v}{\partial n} ds, \tag{2.5}$$

pois, $\nabla^2(u) = \nabla^2(1) = 0$ e $\frac{\partial u}{\partial n} = \frac{\partial 1}{\partial n} = 0$.

Utilizando as seguintes considerações:

- P sendo o centro do cilindro;
- r sendo o raio do cilindro;
- Γ sendo o domínio formado por todos os pontos cujo raio é igual a r.

Pelas equações 2.1 e 2.5

$$\begin{aligned} -\int_\Omega \delta\left(P - Q\right) d\Omega &= -\int_0^{2\pi} \left(\frac{A}{r}\right) r d\theta \\ &= -\int_0^{2\pi} A d\theta \\ &= -2\pi A, \end{aligned} \tag{2.6}$$

onde A é a constante a ser determinada, $r = \rho$ é o raio do cilindro, e θ é o ângulo formado entre a fonte e as extremidades do elemento diferencial de contorno.

Utilizando o fato da função delta de Dirac ser zero para qualquer ponto do domínio exceto na origem, tem-se:

$$\begin{aligned} \int_\Omega \delta(P-Q) &= 1 \\ 2\pi A &= 1 \\ A &= \frac{1}{2\pi}. \end{aligned} \tag{2.7}$$

Substituindo na equação 2.3, a solução fundamental é escrita na forma:

$$v = \frac{1}{2\pi}\ln(r). \tag{2.8}$$

Essa solução também é chamada de função de espaço livre de Green e possui a importante característica de simetria, isto é, $v(Q,P) = v(P,Q)$. Essa propriedade será usada adiante para calcular a integral de contorno.

2.2 EQUAÇÃO INTEGRAL DE CONTORNO

A fim de obter a formulação do método dos elementos de contorno, é necessário relacionar as entidades do domínio com as entidades de contorno, fazendo com que toda a representação fique restrita a este último.

Considerando o caso mais genérico, onde se tem a Equação de Laplace e condições de contorno mistas, tem-se:

$$\nabla^2 u = 0, \tag{2.9}$$

com as seguintes condições de contorno: $u = \bar{u}$ em Γ_1 e $\frac{\partial u}{\partial n} = \bar{u}_n$ em Γ_2, onde $\Gamma_1 \cup \Gamma_2 = \Gamma$ é o contorno total. A notação em barras indica que os valores são conhecidos.

Supondo o ponto fonte $P(x,y)$ localizado em (x,y) e utilizando a segunda identidade de Green (2.4) tem-se:

$$-\int_\Omega u(Q)\delta(Q-P)d\Omega_q = \int_\Gamma \left[v(q,p)\frac{\partial u(q)}{\partial n_q} - u(q)\frac{\partial v(q,P)}{\partial n_q} \right] ds_q, \tag{2.10}$$

onde $P, Q \in \Omega$, $q, p \in \Gamma$, $d\Omega$ é um elemento diferencial do domínio e ds é um elemento diferencial do contorno. Além disso, a notação com subscrito indica em que ponto os elementos são analisados. No caso da equação acima, os elementos são analisados no

ponto q.

Utilizando o princípio da simetria, $v(Q,P) = v(P,Q)$ e a propriedade $\int_\Omega \delta(Q - Q_0)h(Q)d\Omega_q = h(Q_0)$ da função Delta, pode-se reescrever a equação (2.10) da seguinte maneira:

$$u(P) = -\int_\Gamma \left[v(P,q)\frac{\partial u(q)}{\partial n_q} - u(q)\frac{\partial v(P,q)}{\partial n_q} \right] ds_q, \tag{2.11}$$

onde as variáveis v e $\frac{\partial v}{\partial n}$ são conhecidas da seção anterior, pois v é a solução fundamental e $\frac{\partial v}{\partial n}$ sua derivada no ponto q em relação ao vetor normal $\vec{n}$. Reproduzindo os resultados obtidos, tem-se:

$$v = \frac{1}{2\pi}\ln(r), \tag{2.12}$$

$$\frac{\partial v}{\partial n} = \frac{1}{2\pi}\frac{\cos(\phi)}{r}, \tag{2.13}$$

onde $r = |q - P|$ e $\phi =$ ângulo$(\vec{r}, \vec{n})$.

Essas relações, juntamente com a equação 2.11, podem ser utilizadas para calcular a solução u em qualquer ponto interno ao domínio. Porém, a fim de obter a solução em pontos do contorno, é necessário desenvolver uma equação semelhante a 2.11 que remova as singularidades ln(r) de u e $\frac{1}{r}$ de $\frac{\partial u}{\partial n}$.

Suponha um caso geral em que o ponto P esteja localizado em um canto do contorno e que o domínio Ω^* seja o resultado da extração de uma pequena região circular de raio ϵ. Então, a figura 3 será utilizada para obter algumas relações geométricas. No diagrama, Γ_ϵ é um arco circular de extremo nos pontos A, B e P, e também, a soma dos arcos PA e PB é l. Além disso, o vetor normal $\vec{n}$ coincide com ϵ, apontando para o ponto P e o ângulo entre as linhas tangentes ao contorno e o ponto P é α.

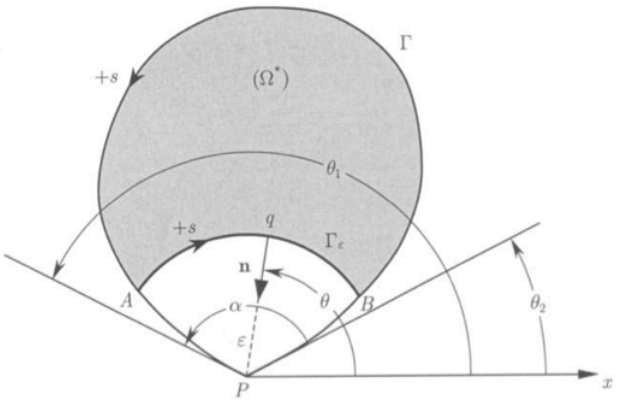


Figura 3: Definições geométricas relacionadas a um ponto P em um contorno não suave [fonte: Katsikadelis(2002)]

Pela geometria apresentada no diagrama, pode-se fazer as seguintes considerações:

$$\lim_{\epsilon\to 0}(\theta_1-\theta_2)=\alpha,$$
$$\lim_{\epsilon\to 0}\Gamma_\epsilon=0,$$
$$\lim_{\epsilon\to 0}\Gamma-l=\Gamma.$$

Fazendo uso da segunda identidade de Green no domínio Ω^* para as funções u e v, e sabendo que para um ponto fora do domínio $\delta(Q-P)=0$, tem-se:

$$\int_{\Omega^*} u\delta(Q-P)d\Omega = 0, \tag{2.14}$$

$$\int_{\Gamma-l}\left(v\frac{\partial u}{\partial n}-u\frac{\partial v}{\partial n}\right)ds+\int_{\Gamma_\epsilon}\left(v\frac{\partial u}{\partial n}-u\frac{\partial v}{\partial n}\right)ds = 0. \tag{2.15}$$

Analisando o comportamento da primeira integral quando $\epsilon\to 0$, isto é, quando o ponto campo se aproxima do ponto fonte tem-se:

$$\lim_{\epsilon\to 0}\int_{\Gamma-l}\left(v\frac{\partial u}{\partial n}-u\frac{\partial u}{\partial n}\right)ds=\int_{\Gamma}\left(v\frac{\partial u}{\partial n}-u\frac{\partial v}{\partial n}\right)ds. \tag{2.16}$$

Analisando o comportamento da segunda integral quando $\epsilon\to 0$ tem-se:

$$\int_{\Gamma_\epsilon}\left(v\frac{\partial u}{\partial n}-u\frac{\partial v}{\partial n}\right)ds=\underbrace{\int_{\Gamma_\epsilon}\frac{1}{2\pi}\frac{\partial u}{\partial n}\ln(r)ds}_{I_1}-\underbrace{\int_{\Gamma_\epsilon}\frac{1}{2\pi}u\frac{\cos(\phi)}{r}ds}_{I_2}. \tag{2.17}$$

Para o arco circular Γ_ϵ, $r = \epsilon$ e $\phi = \pi$, este último devido ao contorno suave. Além disso, $ds = \epsilon(-d\theta)$, pois o ângulo θ é positivo no sentido anti-horário, ou seja, oposto ao incremento ds.

Considerando a integral I_1,

$$\int_{\Gamma_\epsilon} \frac{1}{2\pi}\frac{\partial u}{\partial n}\ln(r)ds = \int_{\theta_1}^{\theta_2} \frac{1}{2\pi}\frac{\partial u}{\partial n}\epsilon\ln(\epsilon)d(-\theta). \tag{2.18}$$

Pelo teorema do valor médio do cálculo diferencial e integral, sabe-se que o valor da integral é igual ao valor de seu integrando em algum ponto O do intervalo, multiplicado pela largura do intervalo. Assim a integral acima pode ser escrita como:

$$I_1 = \frac{1}{2\pi}\left[\frac{\partial u}{\partial n}\right]_O \epsilon\ln(\epsilon)(\theta_1 - \theta_2). \tag{2.19}$$

quando $\epsilon \to 0$, o ponto O se aproxima de P:

$$\lim_{\epsilon\to 0} \epsilon\ln(\epsilon) = I_1 = 0. \tag{2.20}$$

Analogamente, considerando a integral I_2, temos:

$$I_2 = -\int_{\Gamma_\epsilon} \frac{1}{2\pi}u\frac{\cos(\phi)}{r}ds = -\int_{\theta_1}^{\theta_2} \frac{1}{2\pi}u\frac{-1}{\epsilon}\epsilon d(-\theta). \tag{2.21}$$

Pelo teorema do valor médio,

$$I_2 = -\frac{1}{2\pi}u_O(\theta_2 - \theta_1) = \frac{\theta_1 - \theta_2}{2\pi}u_O. \tag{2.22}$$

Tomando o limite quando $\epsilon \to 0$, obtém-se o comportamento de I_2:

$$\lim_{\epsilon\to 0} I_2 = \frac{\alpha}{2\pi}u(P). \tag{2.23}$$

Substituindo esses resultados em $\int_{\Gamma_\epsilon}\left(v\frac{\partial u}{\partial n} - u\frac{\partial v}{\partial n}\right)ds = \int_{\Gamma_\epsilon}\frac{1}{2\pi}\frac{\partial u}{\partial n}\ln(r)ds - \int_{\Gamma_2}\frac{1}{2\pi}u\frac{cos\phi}{r}ds$

Tem-se:

$$lim_{\epsilon\to 0}\int_{\Gamma_\epsilon}\left[v\frac{\partial u}{\partial n} - u\frac{\partial v}{\partial n}\right]ds = \frac{\alpha}{2\pi}u(P). \tag{2.24}$$

Substituindo esse resultado em $0 = \int_{\Gamma - l}\left(v\frac{\partial u}{\partial n} - u\frac{\partial v}{\partial n}\right)ds + \int_{\Gamma_\epsilon}\left(v\frac{\partial u}{\partial n} - u\frac{\partial v}{\partial n}\right)ds$ tem-se:

$$\frac{\alpha}{2\pi}u(p) = -\int_{\Gamma}\left[v(p,q)\frac{\partial u(q)}{\partial n_q} - u(q)\frac{\partial v(p,q)}{\partial n_q}\right]ds_q. \tag{2.25}$$

A equação 2.25 é chamada de BIE da solução da Equação de Laplace, considerando os pontos de contorno não suaves $p \in \Gamma$. Quando suave, então $\alpha = \pi$, e a equação integral de contorno pode ser representada por:

$$\frac{1}{2}u(p) = -\int_{\Gamma}\left[v(p,q)\frac{\partial u(q)}{\partial n_q} - u(q)\frac{\partial v(p,q)}{\partial n_q}\right]ds_q. \tag{2.26}$$

Por outro lado, para um ponto fora do domínio Ω, a segunda identidade de Green leva a:

$$0 = -\int_{\Gamma}\left[v(P,q)\frac{\partial u(q)}{\partial n_q} - u(q)\frac{\partial v(P,q)}{\partial n_q}\right]ds_q. \tag{2.27}$$

Generalizando os resultados obtidos, temos:

$$\epsilon(P)u(P) = -\int_{\Gamma}\left[v(P,q)\frac{\partial u(q)}{\partial n(q)} - u(q)\frac{\partial v(P,q)}{\partial n_q}\right]ds_q. \tag{2.28}$$

Onde $\epsilon(P)$ é um coeficiente que depende da posição do ponto P obedecendo as seguintes situações:

$$\epsilon(P) = \begin{cases} 1 & \text{para } P \in \Omega, \quad (2.29) \\ 1/2 & \text{para } P = p \text{ sob o contorno suave } \Omega, \quad (2.30) \\ 0 & \text{para } P \text{ fora de } \Omega. \quad (2.31) \end{cases}$$

2.3 MODELAGEM NUMÉRICA

A fim de resolver o problema numericamente, é necessário discretizar a geometria e as características conhecidas do problema, dividindo o contorno em pequenos pedaços chamados de elementos. Segue uma representação de um domínio discretizado, dando ênfase a um elemento do contorno.

A discretização pode ser feita através de funções das mais variadas ordens, sendo mais comuns os elementos constantes, lineares (1^a ordem) e quadráticos (2° grau). No presente trabalho, foi adotada a discretização em elementos constantes, para as características conhecidas do problema, ou seja, possui o mesmo valor em todo o elemento.

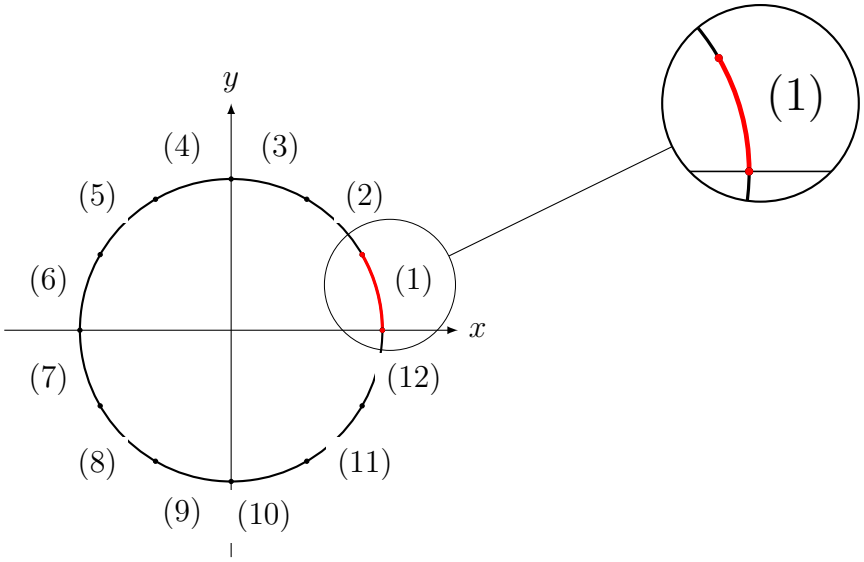


Figura 4: Discretização do contorno.

2.3.1 ELEMENTOS CONSTANTES

Considerando as quantidades u e $\frac{\partial u}{\partial n} = u_n$ constantes em cada elemento, com nó localizado no ponto médio de cada um deles, e supondo que o contorno seja suave, pode-se escrever a equação 2.26 da seguinte maneira:

$$\frac{1}{2}u^i = -\sum_{j=1}^{N}\int_{\Gamma_j} v(p,q)\frac{\partial u(q)}{\partial n_q}ds_q + \sum_{j=1}^{N}\int_{\Gamma_j} u(q)\frac{\partial v(p,q)}{\partial n_q}ds_q, \tag{2.32}$$

onde Γ_j é o elemento no qual o i-ésimo nó está localizado e no qual a integral é calculada e p_i é o ponto nodal do i-ésimo elemento. Como as quantidades u e u_n são constantes, pode-se extraí-las da integral e após uma simples reorganização dos termos da equação tem-se:

$$-\frac{1}{2}u^i + \sum_{j=1}^{N}\left(\int_{\Gamma_j}\frac{\partial v}{\partial n}ds\right)u^j = \sum_{j=1}^{N}\left(\int_{\Gamma_j} v ds\right)u_n^j. \tag{2.33}$$

As integrais da equação acima contribuem diretamente para o valor de $\frac{1}{2}u^i$, recebendo uma nomenclatura especial: coeficientes de influência. Assim,

$$\hat{H}_{ij} = \int_{\Gamma_i}\frac{\partial v(p_i,q_j)}{\partial n_q}ds, \tag{2.34}$$

$$G_{ij} = \int_{\Gamma_i} v(p_i,q_j)ds, \tag{2.35}$$

onde, para um dado elemento, p_i é mantido fixo e q é variável, recebendo os nomes de ponto fonte e ponto campo, respectivamente.

Adicionando a nova nomenclatura à equação 2.33, tem-se

$$-\frac{1}{2}u^i + \sum_{j=1}^{N} \hat{H}_{ij}u^j = \sum_{j=1}^{N} G_{ij}u_n^j. \tag{2.36}$$

Além disso, sabe-se que existe uma singularidade quando o ponto fonte e ponto campo estão no mesmo elemento, devido à quantidade ln(r) da solução fundamental. Então, para os casos em que $i = j$, é necessário adicionar $\frac{1}{2}$ ao resultado do coeficiente de influência $\hat{H}_{ij}$, a fim de remover a singularidade. Então, o coeficiente de influência $\hat{H}_{ij}$ pode ser escrito como:

$$H_{ij} = \hat{H}_{ij} - \frac{1}{2}\delta_{ij}. \tag{2.37}$$

Substituindo na equação 2.36, temos:

$$\sum_{j=1}^{N} H_{ij}u^j = \sum_{j=1}^{N} G_{ij}u_n^j. \tag{2.38}$$

Esta última equação é aplicada a todos os nós $p_i(i = 1, 2, ..., N)$ de cada elemento, formando um sistema de N equações lineares que, em forma matricial, assumem a seguinte forma:

$$[H]u = [G]u_n, \tag{2.39}$$

onde $[H]$ e $[G]$ são matrizes $N \times N$; u e u_n são vetores de N dimensões.

Considerando um problema de condições de contorno mistas tem-se que u é conhecido em parte do contorno e u_n é conhecido em outra parte, ou seja,

$$\Gamma = \Gamma_1 \cup \Gamma 2,$$
$$\bar{u} \text{ em } \Gamma 1,$$
$$\bar{u}_n \text{ em } \Gamma 2.$$

Assim, a equação 2.39 pode ser escrita como:

$$[[H_{11}] + [H_{12}]] \begin{pmatrix} \bar{u}_1 \\ u_2 \end{pmatrix} = [[G_{11}] + [G_{12}]] \begin{pmatrix} u_{n1} \\ \bar{u_{n2}} \end{pmatrix}, \tag{2.40}$$

onde $\bar{u}_1$ e $\bar{u}_{n2}$ são os valores conhecidos de Γ_1 e Γ_2, respectivamente; u_{n1} e u_2 os valores desconhecidos.

Comparando com um sistema linear padrão $[A]\{x\} = \{b\}$, observa-se que $[A]$ irá conter todas as colunas de $[H]$ e $[G]$ que possuírem variáveis desconhecidas u e u_n, enquanto o vetor b será o restante das colunas de $[H]$ e $[G]$. O fato da matriz resultante ser densa facilita a montagem do algoritmo, porém sua resolução é mais cara em termos de processamento e memória.

Realizando a reorganização dos termos, pode-se escrever o sistema de equações da seguinte maneira:

$$\underbrace{[[H_{12}] - [G_{11}]]}_{[A]} \underbrace{\begin{pmatrix} u_2 \\ u_{n1} \end{pmatrix}}_{\{X\}} = \underbrace{[-H_{11}]\{\bar{u}_1\} + [G_{12}]\{\bar{u_{n2}}\}}_{\{B\}}. \tag{2.41}$$

O formato do sistema de equações lineares acima ilustra exatamente a maneira como a rotina de montagem das matrizes $[A]$ e $[B]$ (`buildABMatrix`) reorganiza os termos conhecidos e desconhecidos das matrizes $[G]$ e $[H]$, obtendo, assim, o sistema de equações lineares.

É possível, também, obter a solução u em pontos internos ao contorno Γ, ou seja, o domínio Ω, através da seguinte equação:

$$u(P) = \sum_{j=1}^{N} \hat{H}_{ij} u^j - \sum_{j=1}^{N} G_{ij} u_n^j, \tag{2.42}$$

onde os coeficientes de influência $\hat{H}_{ij}$ e G_{ij} são calculados em algum ponto $P \in \Omega$ pelas integrais $\int_\Gamma \frac{\partial v(p_i,q)}{\partial n_q} ds$ e $\int_{\Gamma_i} v(p_i, q) ds$, respectivamente.

2.3.2 CÁLCULO DAS INTEGRAIS DOS COEFICIENTES DE INFLUÊNCIA

Existem diversas técnicas de integração numérica, como: regra do trapézio, $\frac{1}{3}$ Simpson, quadratura de Gauss, entre outras.

A integração numérica utilizando a regra do trapézio consiste em aproximar a função $f(x)$ por uma equação linear dentro do intervalo de integração $[a, b]$. Após essa aproximação, calcula-se a área abaixo do trapézio, fornecendo uma aproximação do valor de sua integral. Esse método pode ser refinado aumentando o número de trapézios no intervalo, e tomando a aproximação da integral como sendo a soma das áreas dos trapézios fazendo com que o resultado obtido seja mais acurado. Porém, o cálculo se torna mais ineficiente, uma vez que mais operações são necessárias para obter a aproximação.

A integração numérica utilizando a regra $\frac{1}{3}$ Simpson faz uso de um polinômio de Lagrange de segunda ordem para aproximar a função $f(x)$ e, analogamente à regra do trapézio, divide o intervalo em diversos segmentos que, somadas suas áreas, fornecem uma aproximação da integral da função. Assim, como no método dos trapézios, o cálculo das áreas é ineficiente por exigir uma grande quantidade de operações e o resultado final pode não ser satisfatório devido à acurácia, apesar deste método ser mais eficiente do que o primeiro.

Neste trabalho é utilizada a quadratura de Gauss para o cálculo das integrais dos coeficientes de influência, pois, como será mostrado adiante, fornece um excelente balanço acuária vs. eficácia.

O ponto chave da quadratura de Gauss é determinar pontos no intervalo de integração, de forma que os erros positivos e negativos se compensem.

Na figura 5, os pontos vermelhos são os pontos de Gauss. Cada um desses pontos possui um peso, que foi calculado de forma a reduzir o erro e aumentar a acurácia do resultado. A equação utilizada para os cálculos será apresentada adiante.

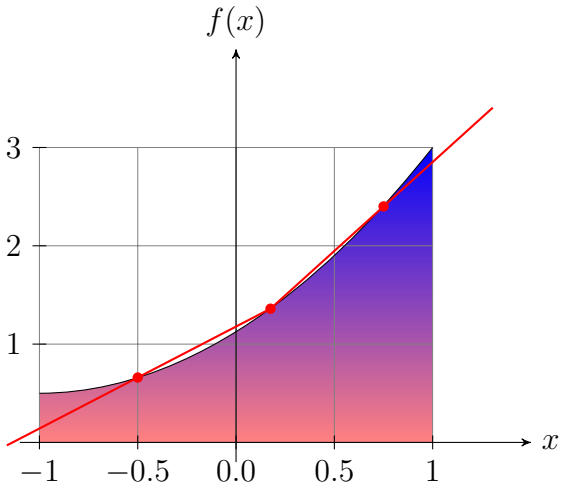


Figura 5: Integração de Gauss

A grande vantagem da quadratura de Gauss em relação aos outros métodos é que ela

utiliza polinômios ortogonais, fazendo com que a representação da função $f(x)$ seja exata para polinômios de grau igual ou inferior a $2k + 1$. Esse fato faz com que o método seja classificado como superacurado.

Para efetuar os cálculos dos coeficientes de influência que surgem do desenvolvimento do método dos elementos de contorno, faz-se necessário distinguir o cálculo quando o ponto campo está longe do campo fonte ($i \neq j$) do cálculo quando o ponto campo está próximo ao campo fonte ($i = j$). Além disso, o intervalo de integração utilizado na quadratura de Gauss é $[-1, 1]$. Assim, é necessário utilizar uma transformação de coordenadas para converter as coordenadas globais para coordenadas locais (interna ao elemento tendo como origem o seu ponto médio).

Como dito anteriomente, a integração através da quadratura de Gauss utiliza uma transformação de coordenadas para que o elemento fique no intervalo $[-1, 1]$. Assim, o seguinte cálculo é realizado:

$$\int_{-1}^{1} f(\xi)d\xi = \sum_{k=1}^{N} w_k f(\xi_k). \tag{2.43}$$

Na equação acima, $f(\xi)$ é a função a ser integrada, N o número de pontos de Gauss e w_k os pesos de Gauss. Os pesos de Gauss já são bem estabelecidos na literatura que aborda métodos numéricos. A tabela com abscissas e pesos de uma função com singularidade logarítmica, fornecida por Katsikadelis (17) e utilizada neste trabalho, pode ser observada no apêndice A.

2.3.2.1 PONTO CAMPO DISTANTE DO PONTO FONTE

Suponha um elemento j, com coordenadas extremas (x_j, y_j) e (x_{j+1}, y_{j+1}) onde a integração será realizada, expressas em coordenadas globais, e origem em um ponto p_i.

Utilizando como base a figura 6 e aplicando os conceitos de geometria analítica, é possível desenvolver um sistema de coordenadas que facilite a resolução do problema.

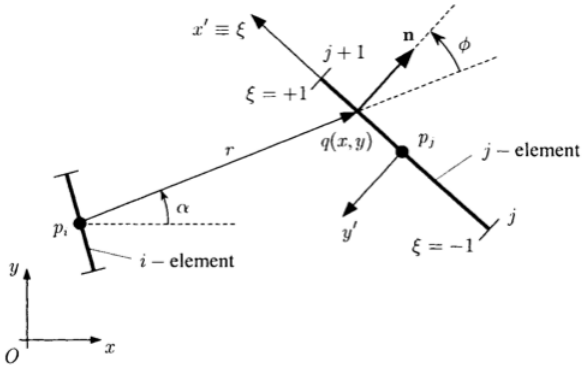


Figura 6: Sistema de coordenadas globais e locais [fonte: Katsikadelis(2002)]

Analisando a figura **??**, pode-se desenvolver um sistema de cordenadas com centro no ponto q, localizado no ponto médio do elemento j e, possuindo eixos coordenados $x^{'}$ e $y^{'}$, as seguintes equações podem ser utilizadas para relacionar a coordenada local $(x^{'}, 0)$ do ponto q no j-ésimo elemento e o sistema de coordenadas globais $x - y$:

$$\begin{aligned}
\xi &= \frac{x^{'}}{\frac{l_j}{2}}, \\
x(\xi) &= \frac{x_{j+1} + x_j}{2} + \frac{x_{j+1} - x_j}{2}\xi, \\
y(\xi) &= \frac{y_{j+1} + y_j}{2} + \frac{y_{j+1} - y_j}{2}\xi, \\
ds &= \sqrt{dx^2 + dy^2} = \sqrt{\frac{x_{j+1} - x_j^2}{2} + \frac{y_{j+1} - y_j^2}{2}} = \frac{l_j}{2}d\xi,
\end{aligned}$$

onde (x_j, y_j) são as coordenadas do início do j-ésimo elemento em coordenadas globais, (x_{j+1}, y_{j+1}) as coordenadas da outra extremidade do elemento, l_j o comprimento do elemento. Neste exemplo, a relação ϵ é utilizada para mapear as coordenadas globais em um intervalo $[-1, 1]$.

Substituindo essas transformações em G_{ij} tem-se

$$G_{ij} = \int_{\Gamma_j} v ds = \int_{-1}^{1} \frac{1}{2\pi} \ln[r(\xi)] \frac{l_j}{2} d\xi = \frac{l_j}{4\pi} \sum_{k=1}^{N} \ln[r(\xi_k)] w_k, \tag{2.44}$$

onde $r(\xi_k) = \sqrt{[x(\xi_k) - x_i]^2 + [y(\xi_k) - y_i]^2}$.

Para o coeficiente $\hat{H}_{ij}$ é mais conveniente utilizar a solução analítica:

$$\begin{aligned}
ds \cdot \cos(\phi) &= r \cdot d\alpha, \\
\hat{H}_{ij} &= \int_{\Gamma_{ji}} \frac{\partial v}{\partial n} ds = \int_{\Gamma_j} \frac{1}{2\pi} \frac{\cos(\phi)}{r} ds = \int_{\Gamma_j} \frac{1}{2\pi} d\alpha = \frac{a_{j+1} - a_j}{2\pi},
\end{aligned}$$

onde (x_j, y_j) e (x_{j+1}, y_{j+1}) são os extremos do j-ésimo elemento e (x_i, y_i) são as coordenadas do ponto fonte P; a_{j+1} e a_j são ângulos calculados por $tan(a_{j+1}) = \frac{y_{j+1}-y_i}{x_{j+1}-x_i}$ e $tan(a_j) = \frac{y_j-y_i}{x_j-x_i}$.

2.3.2.2 PONTO CAMPO PRÓXIMO AO PONTO FONTE

Neste caso, há uma singularidade devido aos fatores $\ln r$ e $\frac{1}{r}$ da solução fundamental. Como ponto fonte e ponto campo pertencem ao mesmo elemento, sendo, portanto, muito próximos, tem-se que $\phi = \frac{\pi}{2}$ ou $\phi = \frac{3\pi}{2}$, ou seja, $cos\phi = 0$. Então,

$$\begin{aligned} x_{p_j} &= \frac{x_{j+1} + x_j}{2}, \\ y_{p_j} &= \frac{y_{j+1} + y_j}{2}, \\ r(\xi) &= \sqrt{[x(\xi) - x_{p_j}]^2 + [y(\xi) - y_{p_j}]^2} = \frac{l_j}{2}|\xi|. \end{aligned}$$

Utilizando as relações acima, pode-se definir G_{jj} da seguinte maneira:

$$G_{jj} = \int_{\Gamma_j} \frac{1}{2\pi}\ln(r)ds = 2\int_0^{\frac{lj}{2}} \frac{1}{2\pi}\ln(r)ds = \frac{1}{\pi}[r \cdot \ln(r) - r]_0^{\frac{lj}{2}} = \frac{1}{\pi}\frac{l_j}{2}[\ln\frac{l_j}{2} - 1], \qquad (2.45)$$

e $\hat{H_{jj}}$ pode ser definida por:

$$\begin{aligned} \hat{H_{jj}} &= \frac{1}{2\pi}\int_{\Gamma_j} \frac{\cos(\phi)}{r}ds = \frac{1}{2\pi}\int_{-1}^{1} \frac{\cos(\phi)}{|\xi|}d\xi \\ &= \frac{2}{2\pi}\left[\cos(\phi) \cdot \ln|\xi|\right]_0^1 = 0. \end{aligned} \qquad (2.46)$$

3 COMPUTAÇÃO MASSIVAMENTE PARALELA

Em computação, o conceito de paralelismo remete aos computadores vetoriais de grande porte, de meados de 1970, que eram capazes de executar uma operação em um *array* unidimensional de dados utilizando apenas uma instrução. Como um exemplo desse tipo de computador pode-se citar o cray-1. Tal conceito se popularizou de tal forma que, atualmente, dificilmente se encontra um computador pessoal que não tire proveito do conceito de paralelismo.

A alta demanda computacional dos cálculos científicos e o repentino desenvolvimento de novas tecnologias em GPUs, motivado pela exigência dos usuários a fim de obter gráficos cada vez mais realistas, fizeram com que as universidades e empresas fabricantes desenvolvessem novas plataformas com o objetivo de aproveitar o poder computacional crescente das unidades gráficas para cálculos de propósito geral, como os cálculos científicos.

A fim de unificar a linguagem de programação entre os diferentes fabricantes de placas de vídeo, foi criado um consórcio com diversas empresas interessadas nesse tipo de mercado, com a finalidade de criar um padrão livre de custo para a programacão em placas gráficas, padrão este chamado OpenCL.

O OpenCL utiliza como base a linguagem de programação C99 (18), amplamente utilizada no universo da computação. Assim, para aqueles que conhecem esta linguagem, a curva de aprendizado é muito suave, precisando apenas conhecer os conceitos de programação paralela e comunicação entre CPU e GPU.

A arquitetura da linguagem é heterogênea, isto é, parte do código é executado sequencialmente e parte é executada de maneira paralela na GPU, sendo chamados de *kernels* os blocos de código que são executados em paralelo.

3.1 TAXONOMIA DE FLYNN

A taxonomia de Flynn é baseada nos fluxos de instruções e dados e define quatro classificações:

- Single Instruction, Single Data Stream (SISD): Nesta classificação estão os dispositivos que não tiram proveito do paralelismo executando, assim, suas instruções ou fluxo de dados de forma sequencial.
- Single Instruction, Multiple Data Stream (SIMD): Nesta classificação estão os dispositivos capazes de aplicar uma única instrução a diversos fluxos de dados simultaneamente. Neste grupo se encontram os grandes processadores vetoriais, os FPGAs (Field-Programmable Gate Array) e as GPUs.
- Multiple Instruction, Single Data Stream (MISD): Nesta classificação estão os dispositivos capazes de aplicar diversas instruções em um mesmo fluxo de dados, geralmente utilizados em sistemas de alta disponibilidade.
- Multiple Instruction, Multiple Data Stream (MIMD): Nesta classificação estão os dispositivos compostos por múltiplas unidades de processamento autônomas processando simultaneamente instruções distintas. Neste grupo encontram-se as tecnologias de processamento distribuído como MPI.

3.2 CONCEITO DE GPU

O modelo de arquitetura em GPU utiliza a classificação SIMD e faz uso de um computador com CPU, também chamado de *host* ou hospedeiro, responsável por alocar os recursos e enviar os trechos de código que serão executados na GPU, chamados de *kernels*. Além disso, cada GPU é um dispositivo computacional composto por diversas unidades computacionais, ou núcleos, que por sua vez, são compostos de elementos de processamento, que são as unidades mais básicas de computação no modelo.

A figura 7 ilustra a forma como os elementos citados acima anterior estão interconectados.

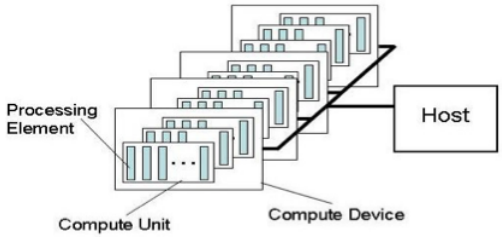


Figura 7: Arquitetura de hardware GPU

3.3 GPU CYPRESS

A arquitetura da placa de vídeo ATI RADEON 5870 (*hardware* utilizado neste trabalho) faz uso do modelo de GPU de codenome Cypress. Esse modelo é compatível tanto com a implementação OpenCL 1.1 quanto com a implementação mais nova, a OpenCL 1.2, utilizada neste trabalho.

Internamente o modelo Cypress é capaz de executar operações atômicas (utilizando apenas 1 ciclo de *clock*) de 32-bits, além de possuir memórias locais de 32KB e memória global compartilhada. O conceito de memórias locais e global é de extrema importância no projeto do algoritmo paralelo, uma vez que, a memória compartilhada local possui uma menor latência, quando comparada com a memória global, sendo recomendável seu uso sempre que possível.

Segue, na figura 8, o diagrama da arquitetura.

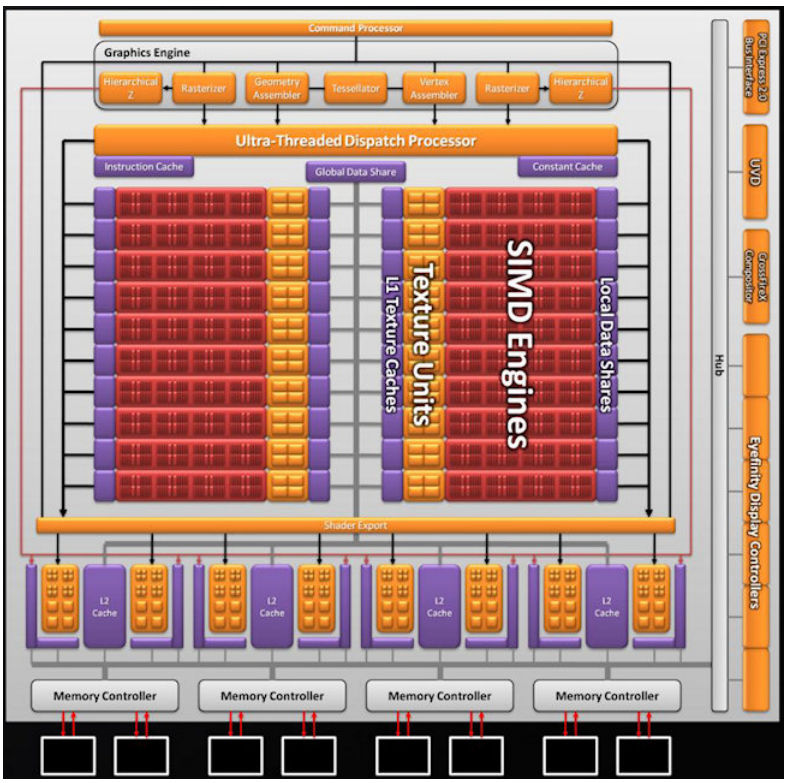


Figura 8: Arquitetura da GPU Cypress

É importante observar a existência de 20 unidades SIMD, onde cada uma é um conjunto de 16 *stream cores*, sendo possível a execução de várias *threads*. Ainda, cada unidade SIMD possui 5 unidades lógico aritméticas (ULA) independentes (sendo uma de uso especial capaz de executar uma adição e multiplicação simultânea em um único ciclo de processador). Porém, podendo compartilhar uma mesma região de memória, já apresentada no parágrafo anterior, chamada de memória local compartilhada.

Assim, pode ser realizado o cálculo que determina a quantidade de threads simultâneas na arquitetura apresentada, como tendo $threads = TPU \cdot ULA = 16 \cdot 4 = 64$ *threads* simultâneas, onde, TPU é quantidade de *stream cores*, ou *thread processing unit*, e ULA a quantidade de unidades lógico aritméticas (excluindo a ULA de uso especial).

O valor teórico de processamento fornecido pela empresa fabricante é de 2,72 Teraflops para operações em precisão simples e 544 Gigaflops em precisão dupla, onde a transferência dos dados entre os componentes é feita através do bus de memória, que possui 153,6 GB/s de largura de banda.

3.4 OPENCL

Com o intuito de aproveitar toda a capacidade computacional de sistemas híbridos (sistemas que possuem CPU e GPUs), foi criada a linguagem de programação de padrão aberto OpenCL.

Scarpino (19) ilustra de uma forma interessante o conceito de concorrência e funcionamento dos diversos componentes da linguagem e do hardware, onde uma analogia é feita utilizando um jogo de cartas de baralho.

Em um jogo de poker, o *dealer* é o responsável por entregar as cartas aos jogadores. Após recebidas, os mesmos analisam suas cartas e decidem as próximas ações. Como são rivais, eles não interagem entre si, apenas fazem requisições ao *dealer*. O *dealer* atende as requisições feitas pelos jogadores e, ao final do jogo, toma o controle de todo o cenário. Seguindo a analogia acima, o *dealer* representa o *host* OpenCL, cada jogador representa um dispositivo ou *device*, o conjunto de cartas representa o contexto, cada uma das cartas representam um *kernel* e as mãos dos jogadores representam a fila de comandos (ou *command queue*).

Nas subseções seguintes, serão explicados brevemente os principais conceitos da linguagem OpenCL. Porém, a abordagem introdutória se baseará apenas nos conceitos que afetam o compreendimento deste trabalho. Para uma abordagem mais ampla, é recomendada a leitura da especificação da linguagem (20).

3.4.1 CONTEXTO

No OpenCL, um contexto identifica um grupo de *devices* que irão trabalhar em conjunto. São os contextos que permitem a criação das *command queues*, que são as estruturas que possibilitam o envio de *kernels* para os *devices*. A linguagem permite a criação de um contexto para cada dispositivo ou um contexto para vários *devices*. Segue o protótipo da função que cria um contexto.

```
1 cl_context clCreateContext (const cl_context_properties *properties,
2                 cl_uint num_devices,
3                 const cl_device_id *devices,
4                 void (*pfn_notify)(const char *errinfo, const void *
                      private_info, size_t cb, void *user_data),
5                 void *user_data,
6                 cl_int *errcode_ret)
```

Os nomes dos argumentos da função acima são auto-explicativos. Dentre eles, podemos destacar:

- `properties`: propriedades do contexto seguindo o padrão <nome> <valor> (as opções disponíveis encontra-se na especificação do OpenCL)
- `num_devices`: número de *devices*;
- `cl_device_id`: identificação do *device*;
- `pfn_notify`: função chamada geralmente para gerenciar erros no contexto;
- `errcode_ret`: variável que irá armazenar o status da criação do contexto.

No algoritmo para uma única GPU, foi criado um contexto e uma *command queue*, anexada apenas ao *device* GPU. O conceito de *command queue* será explicado adiante.

3.4.2 KERNEL

Com tipografia semelhante a uma função em linguagem C, o *kernel* contém o trecho de código que será executado em paralelo, cada *thread* é identificada através da variável `id`, que é obtida, através das funções `get_global_id` e `get_local_id` que especificam a identificação da `thread` globalmente ou dentro de um `workgroup`, respectivamente. Segue um simples exemplo de *kernel* em OpenCL.

```
1 __kernel void add(__global int * a, __global int *b, __global int* c, const
      unsigned n)
2 {
3     int id= get_global_id(0);
4     if (id<n)
5         c[id]= a[id] + b[id];
6 }
```

Segue uma definição dos argumentos e variáveis da função acima:

- `a`: *array* contendo o primeiro operando;
- `b`: *array* contendo o segundo operando;
- `c`: *array* que irá conter o resultado;
- `n`: número de *threads*;

- `id`: identificação global da *thread*;

No exemplo acima, é possível observar que a identificação das *threads* é realizada utilizando a função `get_global_id`. Assim, cada *thread* irá realizar as operações de soma nos *arrays* passados como parâmetros para o *kernel*. É importante observar que toda declaração de variável realizada dentro de um *kernel* OpenCL irá declarar a variável na memória local, assunto abordado no próximo tópico.

A criação do *kernel* no *host* é feita utilizando a função `clCreateKernel`. Segue o protótipo da função.

```
1 cl_kernel clCreateKernel (cl_program program,
2         const char *kernel_name,
3         cl_int *errcode_ret)
```

Onde os argumentos são:

- `program`: programa que contém o *kernel*;
- `kernel_name`: o nome do *kernel* declarado no arquivo de *kernel*;
- `errcode_ret`: variável para armazenamento do status da criação do *kernel*.

Para passar parâmetros para o *kernel*, usa-se a função `clSetKernelArg`. Segue o protótipo da função.

```
1 cl_int  clSetKernelArg (cl_kernel kernel,
2           cl_uint arg_index,
3           size_t arg_size,
4           const void *arg_value)
```

Onde os argumentos são:

- `kernel`: nome do *kernel* que receberá o argumento;
- `arg_index`: inteiro que especifica qual argumento está sendo passado (primeiro, segundo, terceiro, etc.);
- `arg_size`: tamanho do argumento em bytes;
- `arg_value`: valor do argumento.

A chamada do *kernel*, após a passagem de argumentos é feita utilizando a função `clEnqueueNDRangeKernel`. Segue o protótipo da função.

```
1 cl_int  clEnqueueNDRangeKernel (cl_command_queue command_queue,
2                                 cl_kernel kernel,
3                                 cl_uint  work_dim,
4                                 const size_t *global_work_offset,
5                                 const size_t *global_work_size,
6                                 const size_t *local_work_size,
7                                 cl_uint num_events_in_wait_list,
8                                 const cl_event *event_wait_list,
9                                 cl_event *event)
```

- `command_queue`: especifica a fila de comando;
- `kernel`: especifica o *kernel*;
- `workdim`: define as dimensões dos *workgroups* (1D, 2D ou 3D);
- `global_work_offset`: define o tamanho do *offset*;
- `global_work_size`: quantidade de *workitens*;
- `local_work_size`: quantidade de *workitens* por *workgroup*;
- `num_events_in_wait_list`: sinaliza a quantidade de eventos esperada;
- `event_wait_list`: especifica os eventos que precisam ser disparados antes do comando se executado;
- `event`: retorna um objeto evento que identifica a execução do *kernel*.

3.4.3 COMMAND QUEUE

A *command queue* é uma estrutura que recebe as instruções que serão executadas especificando qual dispositivo e contexto serão responsáveis pela execução.

No presente trabalho, foram criadas duas *command queues* (uma para cada *device*), a fim de obter o paralelismo na execução de múltiplas GPUs. Segue o protótipo da função que cria uma *command queue*.

```
1 cl_command_queue clCreateCommandQueue (cl_context context,
2                  cl_device_id device,
```

```
3                   cl_command_queue_properties properties,
4                   cl_int *errcode_ret)
```

Os argumentos da função acima são:

- `context`: especifica o contexto que abrigará a fila;
- `device`: especifica o *device* que a fila pertence;
- `properties`: propriedades da lista de comandos, podendo ser:

  CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE e

  CL_QUEUE_PROFILING_ENABLE. No primeiro caso, especifica se os comandos são executados em ordem ou fora de ordem; no segundo caso, especifica se ocorrerá *profiling* de comandos, ou seja, análise do fluxo de comandos.
- `errcode_ret`: inteiro que armazena o status da criação da lista de comandos.

3.4.4 GERENCIAMENTO DE MEMÓRIA

O gerenciamento de memória é o conceito mais crítico da linguagem, uma vez que, a cópia de memória entre *host* e *device* é a operação que demanda mais ciclos de clock tendo um grande impacto no tempo de execução do programa. Sendo assim, a idéia central ao se utilizar arquiteturas massivamente paralelas é minimizar ao máximo a troca de informações entre *host* e *device*.

A memória da arquitetura utilizada neste trabalho está hierarquizada de acordo com o diagrama da figura 9.

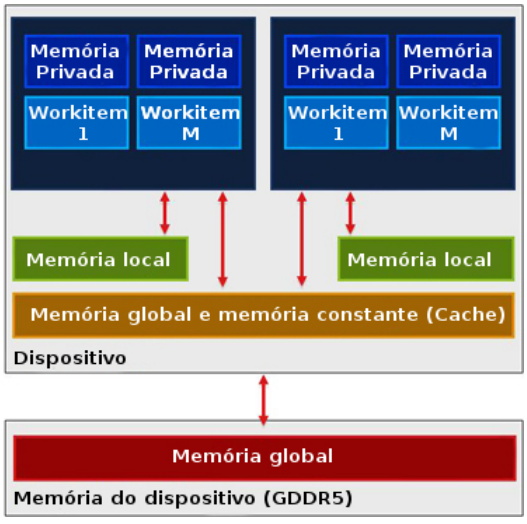


Figura 9: Hierarquia de memória no OpenCL

O modelo acima ilustra a relação entre os diversos tipos de memória dentro de um mesmo *device*. As características de cada um deles são descritas abaixo.

- Memória privada: Similar aos registradores em sistemas CPU, a memória privada na arquitetura OpenCL é compartilhada entre as threads que estão agrupadas em um mesmo *workgroup*. A vantagem do uso nesse tipo de memória é a baixa latência. Apesar dessa vantagem, seu tamanho é reduzido.

  A declaração de tipos de dados nesta região de memória é feita utilizando o identificador __private.

- Memória local: A memória local possui baixíssima latência e geralmente é projetada no próprio chip, fazendo com que seu tamanho seja reduzido. Geralmente, esse tipo de memória é utilizado para possibilitar acesso coalescente entre *workitens* de um mesmo *workgroup*, ou seja, servindo basicamente como uma memória *cache*, evitando o acesso as memórias de maior latência. Observa-se que todas as *threads* de um mesmo *workgroup* compartilham os dados contidos neste tipo de memória.

  A declaração de tipos de dados nesta região de memória é feita utilizando o identificador __local.

- Memória constante: Este tipo de memória é utilizado apenas para leitura no *device*. Como o proprio nome diz, os dados ali contidos são constantes. Na arquitetura AMD ATI, é uma região da memória global (alta latência), com otimizações a fim de obter melhor desempenho em dados de *broadcast*.

  A declaração de tipos de dados nesta região de memória é feita utilizando o identificador __constant.

- Memória global: Geralmente é a memória com maior capacidade no *device*, frequentemente medida na unidade Gigabytes. Apesar de possuir a maior região de armazenamento no dispositivo e ser visível por todas as *threads* na *GPU*, possui alta latência, sendo considerada a maior rival do alto desempenho na arquitetura, tendo seu uso, por vezes, limitado.

  Observa-se que um maior desempenho é obtido quando o acesso à esse tipo de memória é feito em blocos, sendo esse tipo de acesso chamado de *acesso coalescente* (ou *coalesced access*). As otimizações relacionadas ao acesso coalescente variam dependendo do modelo de GPU utilizado; sendo assim, o recomendável é verificar nas especificações do fabricante.

A declaração de tipos de dados nesta região de memória é feita utilizando o identificador __global.

3.4.5 CÓPIA DE DADO DA MEMÓRIA

A cópia de dados entre *host* e *device* é realizada utilizando o comando `clCreateBuffer`. Segue o protótipo da função de cópia de memória.

```
1 cl_mem clCreateBuffer (cl_context context,
2           cl_mem_flags flags,
3           size_t size,
4           void *host_ptr,
5           cl_int *errcode_ret)
```

Os argumentos da função acima são:

- `context`: especifica o contexto onde será realizada a cópia;
- `flags`: especifica como será feito o uso da memória;
  - `CL_MEM_READ_WRITE`: leitura e escrita;
  - `CL_MEM_ONLY_READ`: somente leitura;
  - `CL_MEM_ONLY_WRITE`: somente escrita;
  - `CL_MEM_USE_HOST_PTR`: utilizar a região de memória referenciada pelo host_ptr como armazenamento do objeto memória;
  - `CL_MEM_ALLOC_HOST_PTR`: aloca memória acessível no *host*;
  - `CL_MEM_COPY_HOST_PTR`: aloca a memória para o objeto memória e copia os dados da memória referenciados por `host_ptr`.
- `size`: tamanho da cópia em bytes;
- `host_ptr`: o endereço do *array* alocado no *host*;
- `errcode_ret`: retorna o status da criação do *buffer*.

3.4.6 WORKGROUPS

O paralelismo inerente a linguagem OpenCL consiste da distribuição de tarefas, através dos *kernels*. Isto é, o *kernel* contém a rotina a ser paralelizada e, ao ser chamado,

distribui as instruções entre grupos de *threads*. Surge assim, o conceito de *workgroups* que pode ser definido como grupo de *threads* que possuem memória compartilhada e podem colaborar entre si. Uma observação que deve ser feita é que não é permitida a colaboração entre *threads* que se localizam em *workgroups* distintos.

A divisão de *workgroups* é feita através da chamada do *kernel*, utilizando as variáveis `globalWorkSize` e `localWorkSize`. Segue um diagrama esquemático que mostra a maneira como um *kernel* bidimensional é organizado no *hardware*.

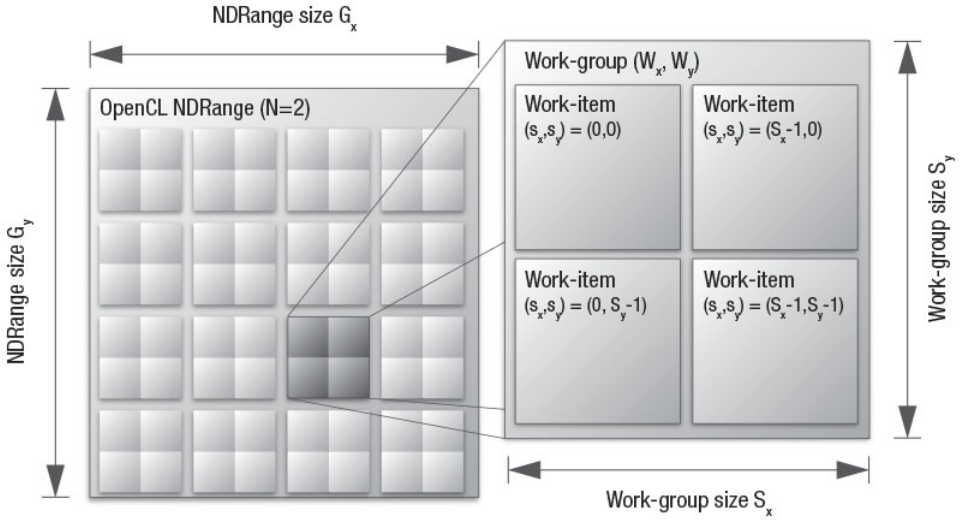


Figura 10: Grid do *kernel*.

A variável `globalWorkSize` define o tamanho do maior grid do *kernel*, ou seja, o NDRange Size, que no diagrama está definido como 8. Além do NDRange, é possível subdividir o grid em *workgroups*, através da variável `localWorkSize`, no diagrama definido como 2. É importante enfatizar que a implementação OpenCL e o *hardware* permite a criação de *grids* tridimensionais, fornecendo, inclusive, tipos de dados específicos para a manipulação de dados em três dimensões, como `float3` e `int3`, porém, o caso 3D está fora do escopo deste trabalho.

A localização da *thread* no maior grid do *kernel* é definida através da função `get_global_id`, enquanto que a localização dentro de um *workgroup* é definida através da função `get_local_id`.

É importante enfatizar que a escolha das dimensões e subdivisões do *kernel* influenciam diretamente o seu desempenho, como será visto adiante. Segundo a especificação da fabricante AMD, o valor de `localWorkSize` que otimiza a execução do *kernel* é 64. Porém, dependendo da aplicação, utilizar esse valor para `localWorkSize` pode ser impraticável devido à questão da colaboração entre as *threads*. Caso seja necessária a colaboração de um grande número de *threads* (mais de 64), o caso ótimo será um número múltiplo de 64.

4 IMPLEMENTAÇÃO

O problema consiste na resolução da Equação de Laplace em duas dimensões, utilizando o desenvolvimento em elementos de contorno apresentado anteriormente.

A fim de validar os resultados, foi utilizado o problema ilustrado na figura 11, que apresenta uma caixa bidimensional de uma unidade de comprimento em cada lateral, onde se conhece o potencial na face esquerda e direita e se conhece a derivada do potencial na face inferior e superior. O problema também inclui um cilindro circular de raio 0,1 de unidade de comprimento, com derivada de potencial conhecida, no centro da caixa. Seguindo a notação adotada no capítulo 2 que expôs o método dos elementos de contorno, as variáveis com valores conhecidos são identificadas com uma barra superior sobre a variável enquanto as variáveis com valores desconhecidos não possuem a barra.

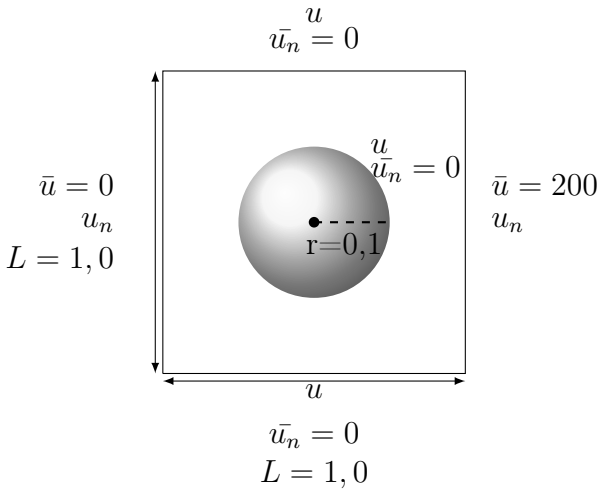


Figura 11: Problema potencial implementado.

O sistema operacional adotado na implementação foi o Gnu/Linux OpenSUSE 12.2, com a implementação OpenCL 1.2 e driver ATI Catalyst 13.1 x86_64 (AMD industries, Sunnyvale, E.U.A), instalados em um servidor Supermicro HD8Ai+ (Super micro computer, San Jose, E.U.A) com 2 processadores Opteron 2350 2Ghz de 4 núcleos, obtendo um total de 8 núcleos de processamento, 4GB de memória RAM DDR3 e duas placas de

vídeo ATI RADEON 5870 1GB GDDR5 (AMD industries, Sunnyvale, EUA).

4.1 IMPLEMENTAÇÃO SERIAL

A implementação numérica serial foi realizada para comparação do tempo de execução com a implementação numérica paralela. A linguagem de programação utilizada foi a C, por ser amplamente utilizada no universo da computação e muito semelhante à linguagem massivamente paralela utilizada neste trabalho.

O algoritmo de alto desempenho escrito permite as configurações dos seguintes parâmetros: número de contornos do programa, número do elemento que finaliza cada um dos contornos, número de nós do contorno e condições de contorno.

Os parâmetros de entrada descritos acima são obtidos através da linha de comando, sendo apresentado ao usuário o que se espera como valor de entrada. É importante ressaltar que a implementação realizada suporta apenas sistemas operacionais que seguem o padrão POSIX (21), ou seja, sistemas como Gnu/Linux e MacOSX.

A fim de tornar o programa modular, foram criadas diversas funções específicas para cada etapa de processamento. As funcionalidades de cada uma delas são apresentadas a seguir.

O diagrama da figura 12 mostra como as diversas funções do programa estão organizadas. A função principal do programa chama-se `main`. Esta função é responsável por definir as variáveis globais da simulação, obtendo os valores através da interação com usuário, e fazer a interface entre as outras funções, recebendo e repassando dados.

A função `preprocessador` é utilizada para discretizar as bordas e definir as condições de contorno do problema de acordo com as entradas do usuário obtidas na função main.

A função `buildGMatrix` recebe as matrizes de coordenadas, calcula os nós geométricos de cada segmento discretizado do contorno, através do ponto médio de cada elemento, e calcula os coeficientes de influência $G_{ij} = v(p_i, q_j)ds$.

A função `buildHMatrix` recebe as matrizes de coordenadas, calcula os nós geométricos de cada segmento discretizado do contorno, através do ponto médio de cada elemento, e calcula os coeficientes de influência $\hat{H_{ij}} = \frac{\partial v(p_i,q_j)}{\partial n_q} ds$ ou ainda, $H_{ij} = \hat{H_{ij}} + \frac{1}{2}\delta_{ij}$.

A função `buildABMatrix` recebe as matrizes dos coeficientes de influência, reorganiza os termos conhecidos e desconhecidos no problema e monta o sistema de equações lineares,

que possui como solução os valores desconhecidos do problema no contorno.

Após a montagem do sistema de equações lineares realizada na etapa anterior, o programa chama a função `leqs` que utiliza o método iterativo de Jacobi para a resolução do sistema, retornando para o programa principal a solução no *array* $x[\,]$.

De posse da solução do sistema, é preciso distinguir que tipo de variável foi resolvida para cada entrada da matriz, ou seja, se foi resolvido u (representado no diagrama como UB) ou u_n (representado no diagrama como UNB). Assim, é chamada a função `Reorder` que faz uma organização desses dados a fim de apresentar ao usuário u e u_n.

A função `Uinter` recebe os resultados calculados na etapa anterior e calcula u e u_n para os pontos internos ao domínio, armazenando os resultados em uma matriz.

Por fim, a função `Output` apresenta as coordenadas dos nós geométricos, solução de u e u_n para o contorno, coordenadas dos pontos internos e a solução de u e u_n nos pontos internos.

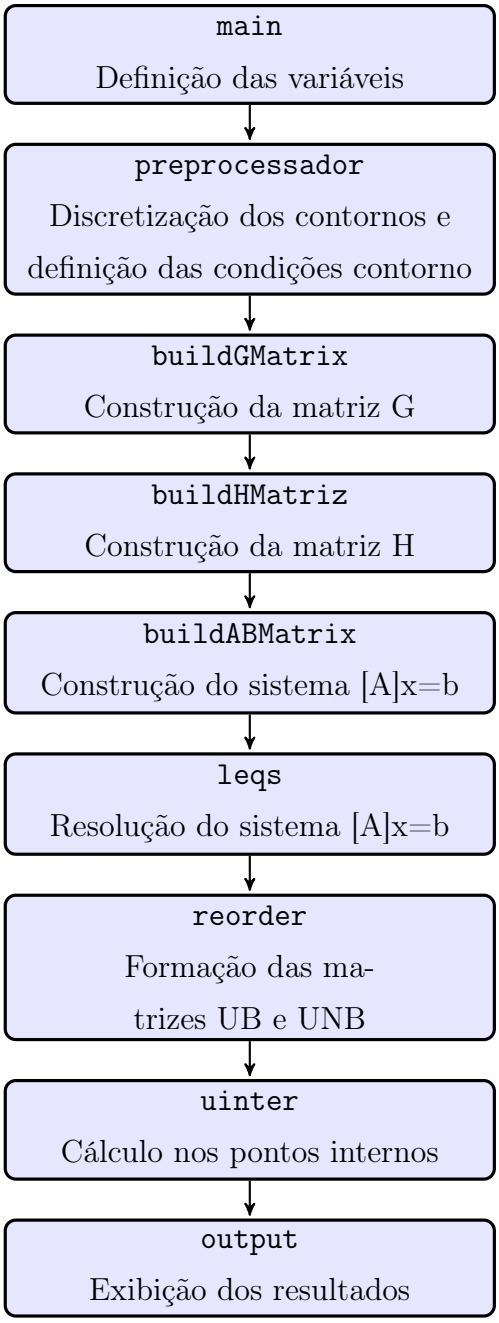


Figura 12: Fluxograma do código serial

Com o objetivo de validar o algoritmo implementado, foi realizada uma execução do algoritmo com 1024 nós de discretização do contorno e o valor do parâmetro de convergência `sum` foi comparado a cada iteração, sendo obtido o gráfico mostrado na figura 13. A curva suave mostra que, a cada iteração, o algoritmo está mais próximo do valor tolerância, ou seja, está convergindo. Esse resultado valida a solução do problema, isto é, o algoritmo cumpre a tarefa imposta a ele que é a resolução do sistema linear com precisão de $1e-10$. Realizando o mesmo teste para diferentes números de nós e nas arquiteturas serial e *multicore*, o resultado é muito semelhante.

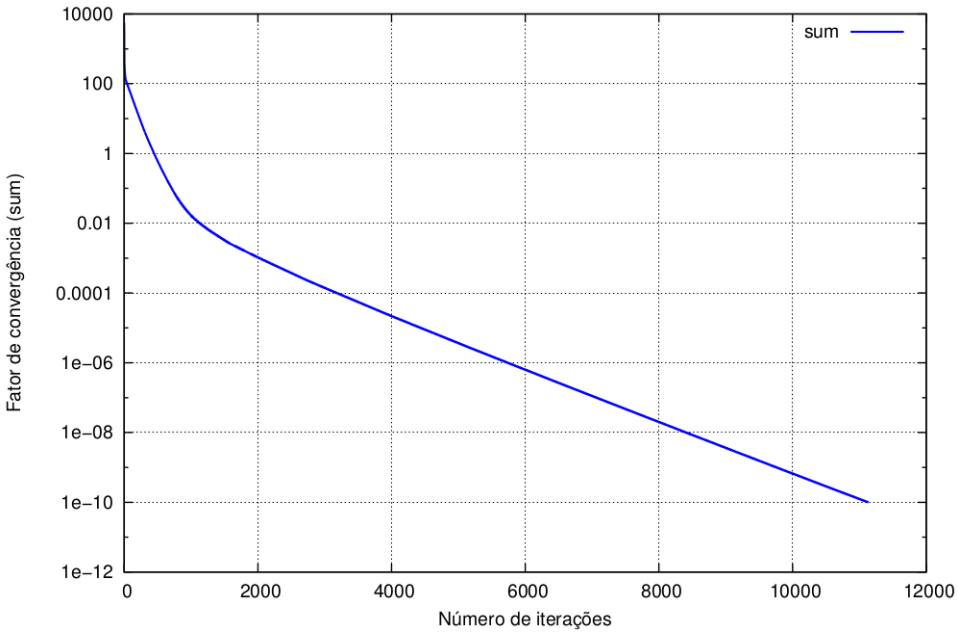


Figura 13: Parâmetro de convergência pela iteração.

A fim de identificar o custo de processamento de cada uma das rotinas acima, foram analisados os tempos de processamento de cada uma delas variando o número de nós. Para cada caso, foram realizadas 10 simulações e a média é apresentada no gráfico apresentado na figura 14 no qual percebe-se que as duas rotinas que mais demandam tempo são as responsáveis pela montagem e resolução do sistema linear, a primeira cresce na ordem de $O(n^{2.3})$ enquanto a última cresce na ordem de $O(n^2)$, segundo a notação assintótica ou notação de Landau..

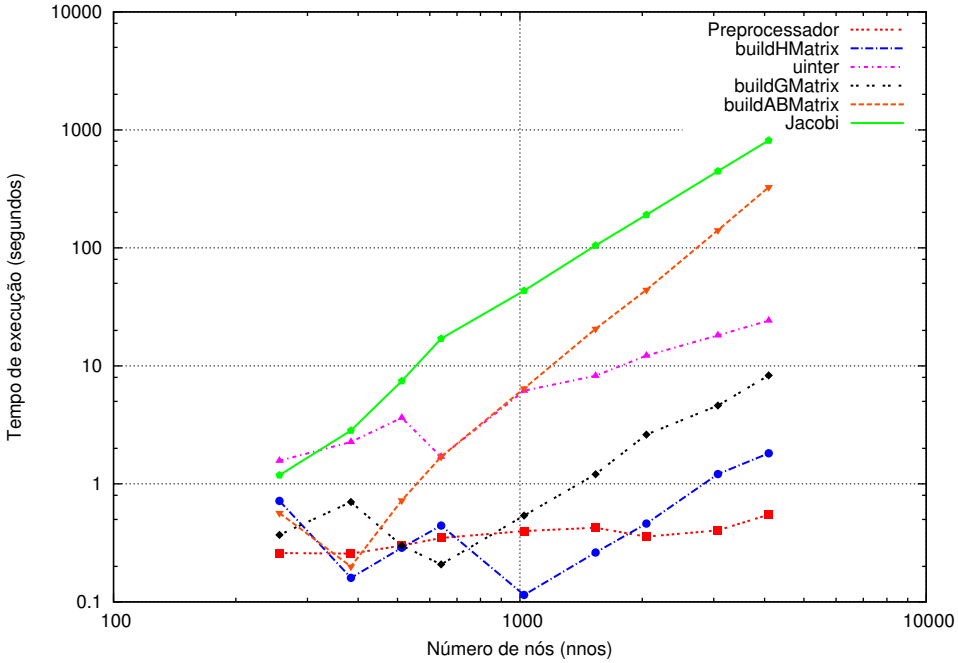


Figura 14: Tempo de execução das rotinas

Ao iniciar o estudo da metodologia e do algoritmo numérico que seria utilizado na resolução da Equação de Laplace, foi levantada a hipótese de que a rotina de resolução de sistemas lineares, através do método iterativo de Jacobi, seria a mais custosa computacionalmente, sendo que as outras rotinas poderiam se manter na forma serial. Porém, como mostram os resultados apresentados acima, foi verificado que a rotina `buildABMatrix` demanda mais processamento do que a rotina de resolução de equações lineares, sugerindo sua paralelização.

Como, em um primeiro momento, o foco do trabalho foi obter uma otimização da rotina de resolução de sistemas lineares, a rotina **buildABMatrix** foi mantida de forma serial. Assim, para uma futura continuidade do trabalho, sugere-se a paralelização dessa rotina.

A seguir, é apresentada a rotina que soluciona o sistema de equações lineares através do algoritmo iterativo de Jacobi. Essa rotina cresce com ordem $O(N^2)$, ou seja, possui um custo computacional extremamente elevado. Sendo assim, sua paralelização é o ponto chave para a otimização do programa.

4.1.1 MÉTODO ITERATIVO DE JACOBI

O método iterativo de Jacobi é o ponto chave para o desempenho da biblioteca, uma vez que resolve sistemas de equações lineares de forma iterativa, demandando alto custo computacional. Suponha um sistema de equações lineares nXn, pode-se escrevê-lo na forma matricial, da seguinte maneira:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{4.1}$$

A matriz $[A]$ pode ser decomposta em duas matrizes, uma matriz diagonal ($[D]$) e a matriz restante ($[R]$), de modo que, $A = D + R$. Assim,

$$\left[\underbrace{\begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}}_{D} + \underbrace{\begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}}_{R}\right] \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}. \tag{4.2}$$

Desta maneira, a solução pode ser obtida iterativamente através da equação:

$$x^{(k+1)} = D^{-1}(b - Rx^{(k)}). \tag{4.3}$$

Para cada elemento da iteração, utiliza-se o algoritmo:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right). \tag{4.4}$$

No algoritmo acima, o somatório pode ser escrito como um laço percorrendo todos os elementos não diagonais da matriz $[A]$, ou seja, todos os elementos da matriz $[R]$. Este somatório é de suma importância para o algoritmo, pois mostra uma dependência dos valores calculados para x nas iterações anteriores. Outra observação importante de otimização no algoritmo é a utilização de um *kernel* para a execução da multiplicação matricial em paralelo.

É importante ressaltar que, ao implementar o algoritmo acima, deve haver uma verificação realizada a cada iteração, a fim de determinar a convergência da solução através de um valor de tolerância, definido, neste trabalho, como $1e^{-10}$. Esse teste de convergência é realizado comparando a soma dos módulos das diferenças entre as iterações, isto é, comparando $sum = \frac{1}{n} \sum |x^{(k+1)} - x^k|$ com $1e^{-10}$ a cada iteração.

4.2 IMPLEMENTAÇÃO PARALELA

Tomando como ponto chave o fato do algoritmo iterativo de Jacobi calcular o valor da solução x com base apenas nos valores já calculados anteriormente, é possível distribuir o cálculo desses valores entre diversos núcleos ou dispositivos computacionais.

Existem diversas ferramentas que possibilitam a paralelização de uma rotina, sendo as mais comuns na comunidade científica: OpenMP, MPI (*Message Passing Interface*) e GPU (*Graphics Processing Unit*). Segue uma breve descrição de cada uma delas.

4.2.1 OPENMP

Segundo a própria comunidade OpenMP, o projeto consiste de uma *Application Programming Interface* (API) para programação em sistemas de memória compartilhada, portável para as plataformas *Unix* e *Windows*, além de suportada pelas linguagens de programação C/C++ e Fortran (22).

A vantagem do uso desta API é a curva de aprendizado que é muito suave, fazendo com que as aplicações possam ser paralelizadas rapidamente. A grande desvantagem é inerente a própria arquitetura da API, pois foi projetada para uso apenas em ambientes de memória compartilhada, sendo possível a paralelização em ambientes de memória distribuída apenas com o uso de outras ferramentas, como o MPI.

4.2.2 MESSAGE PASSING INTERFACE (MPI)

O MPI, como próprio nome diz, é um padrão de passagem de mensagem utilizado na comunicação entre processos remotos. Esse padrão é muito utilizado no meio científico para a paralização de simulações computacionais onde se tem um sistema de memória distribuída.

Apesar de existir uma extensa lista de mais de 300 funções no padrão, a maioria dos programas pode ser paralelizado utilizando apenas 20 delas. O conceito do padrão utiliza um processo pai que cria diversos processos filhos, em computadores distintos, onde são executados os cálculos e repassados ao processo pai para a apresentação ao usuário (23).

A grande vantagem do MPI é a possibilidade de execução de programas em sistemas com memória distribuída. Incluídos nessa categoria estão os *clusters* de computadores, que são um grande agregado de servidores interligados em rede, com a finalidade de aumentar o poder de processamento ou disponibilidade de maneira colaborativa. Entretanto, o MPI possui uma grande dependência na capacidade de transferência da rede, sendo fator limitante no desempenho total da aplicação.

4.2.3 GRAPHICS PROCESSING UNIT (GPU)

Para a resolução de sistemas lineares, foi realizada a paralelização do método de Jacobi distribuindo o cálculo do `x` entre os *stream processors* da GPU, pois, como foi discutido anteriormente, o método depende apenas de valores já calculados de `x`, como foi realizado em (24).

A paralelização citada no parágrafo anterior pode ser definida como uma paralelização trivial, isto é, é feita a distruição dos cálculos sem que seja utilizado nenhum artifício de otimização de acessos à memória ou uso de memória locais através da divisão do grid da memória global em subgrids. Esse tipo de paralelização é típico de problemas onde uma variável não depende da condição da outra variável, como no problema de múltiplos corpos, ou N-Body (25).

Assim, segue o modelo de algoritmo paralelo desenvolvido (15), juntamente com a transcrição do *kernel* utilizado. O algoritmo é executado de maneira análoga ao algoritmo serial explicado anteriormente, até que a rotina Jacobi seja executada. Chegando nessa rotina, é feita uma solicitação de acesso ao *device* pelo *host*, e, caso esteja disponível, o *host* dimensiona o *kernel* e o envia ao *device*, onde o sistema de equações lineares é resolvido de forma massivamente paralela. Após a execução dos cálculos, o *device* envia os resultados na forma de uma matriz, que é armazenada no *host*, passando para o passo seguinte que é a reorganização das matrizes, como explicado anteriormente.

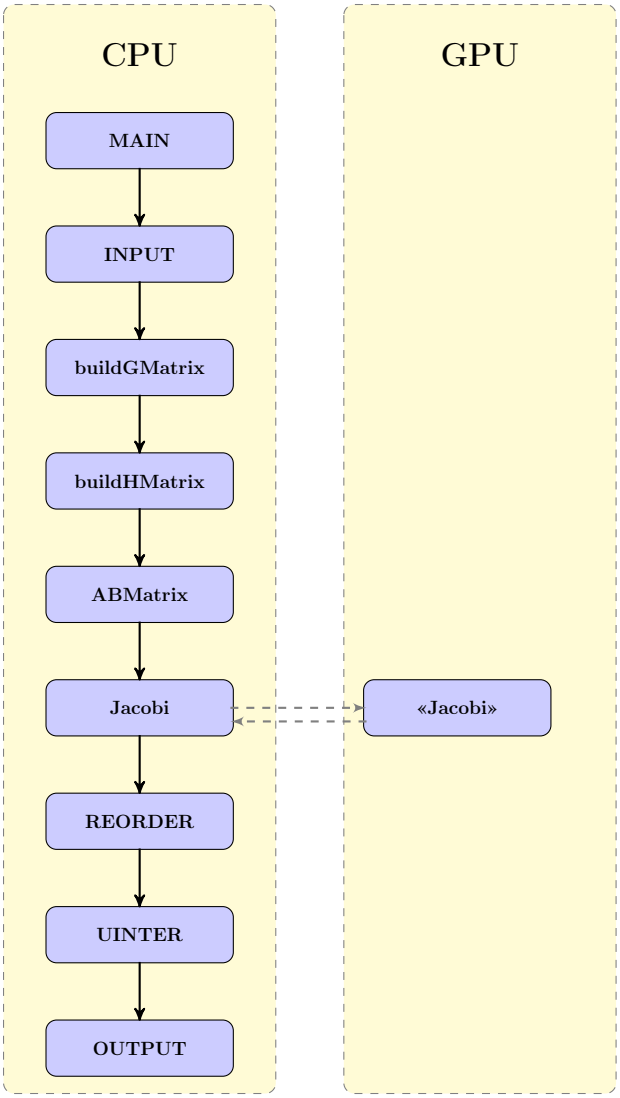


Figura 15: Fluxograma do código paralelo

```
1  __kernel void jacobi_krn (__global double *A, __global double *b,
2  __global double *x,__global double *sum,int nnos) {
3  // Each threads blocks solves for the line indexed by gdx
4      __private int c1,index;
5      __private int ldx  = get_local_id(0);
6      __private int gdx  = get_group_id(0);
7      __private int lsz  = get_local_size(0);
8      __private double atemp,xtemp,stemp;
9      __local double alocal[NNOS];
10
11     stemp = 0;
12     for (c1 = 0; c1 < (nnos/lsz);c1++) {
13      index = c1*lsz+ldx;
14      xtemp = x[index];
15      atemp = A[gdx*NNOS+index];
```

```
16      stemp = mad(atemp,xtemp,stemp);
17    }
18
19    alocal[ldx] = stemp; // Each thread stores its results in local memory
20
21    barrier(CLK_LOCAL_MEM_FENCE);
22
23    stemp=0;
24    if (ldx==0) // This is the master thread {
25     for (c1=0;c1<lsz;c1++)
26       stemp -= alocal[c1];
27       stemp += b[gdx];
28       stemp = stemp/A[gdx*NNOS+gdx];
29       sum[gdx]=stemp; // Stores temporary result in sum
30    }
31 }
```

Segue o fluxograma de interação entre *host* e *device* para o tratamento da rotina de resolução do sistema linear.

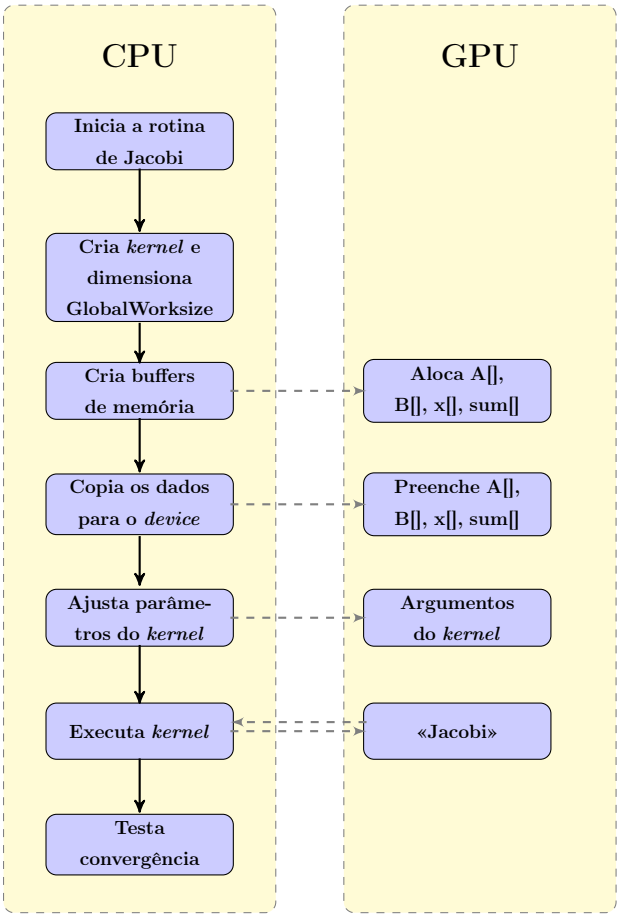


Figura 16: Fluxograma de interação *host* - *device*

Para o caso de uma única GPU, a rotina de resolução de sistema linear utiliza as funções da linguagem OpenCL `clCreateKernel`, `clCreateBuffer`, `clEnqueueWriteBuffer`, `clSetKernelArg`, `clEnqueueNDRangeKernel` e `clEnqueueReadBuffer`.

Primeiramente, a rotina cria o *kernel* e através de um laço dimensiona a variável `globalWorkSize`, somando o valor de `localWorkSize` (igual a 64) a cada iteração, enquanto `globalWorkSize` for menor do que a quantidade de nós do contorno, com a finalidade de manter o valor de `globalWorkSize` sempre múltiplo de `localWorkSize`. Esse artifício otimiza a execução do *kernel*.

A rotina então aloca os *buffers*, através da função `clCreateBuffer`. Na memória global do *device* para leitura, são alocados os vetores `A`, `B`, `x` e `sum`. Este último é responsável pelo teste de convergência. Após a locação, é realizada a cópia dos dados de cada um dos referidos vetores do *host* para o *device* através da função clEnqueueWriteBuffer.

Com os dados contidos no *device*, o *host* ajusta os argumentos do *kernel*, através da rotina `clSetKernelArg`. Então, com as informações necessárias pelo *kernel*, o *host* executa a função `clEnqueueNDRangeKernel` que realiza a chamada do *kernel* criado.

Após a execução do *kernel*, o *device* passa a ter em sua memória os valores calculados. Assim, o *host* realiza uma leitura da memória global do *device*, através da função `clEnqueueReadBuffer`, e armazena os valores em sua memória. Então, utilizando o vetor `sum[ ]`, realiza o teste de convergência da solução.

4.2.4 DETERMINAÇÃO DO PARÂMETRO *WORK-ITEM-SIZE*

Como dito anteriormente, o valor de *localWorkSize* otimiza ou degrada a execução do *kernel*. Com a finalidade de encontrar o valor que otimiza a execução do *kernel*, foram realizadas execuções do algoritmo proposto variando esse valor e verificando o tempo de execução para os casos com 1024 e 4096 nós na borda.

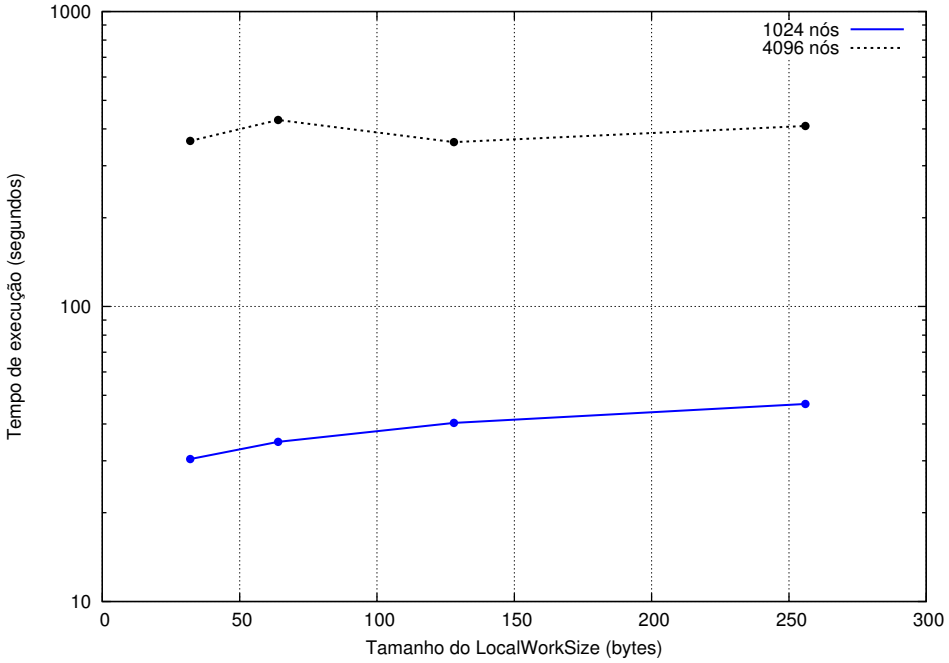


Figura 17: Tamanho do LocalWorkSize pelo tempo de execução do algoritmo de Jacobi

O gráfico acima mostra que apesar do valor sugerido pela fabricante AMD para o `localWorkSize` ser 64, no algoritmo de Jacobi apresentado este valor é 32, que é o valor definido como o estado ótimo da fabricante NVIDIA. Porém, a fim de manter o algoritmo compatível com as diferentes modelos de placas de vídeo foi utilizo o padrão sugerido que é de 64.

Na referência (26), é apresentada uma solução para multiplicação de matrizes cheias no mesmo *hardware* que foi utilizado neste trabalho. Nessa implementação, o autor, mostra que o valor ótimo de *localWorkSize* para o caso de multiplicação de matrizes cheias é 16. Assim, pode-se concluir que, ao realizar uma abordagem de resolução de problemas de forma numérica em GPU, uma atenção deve ser dada à essa variável, pois afeta diretamente o tempo de computação da GPU, sendo recomendável testes com os diferentes valores a fim de identificar o valor que oferece o melhor desempenho para o problema analisado.

Segundo a referência (27), o aumento da latência ocorrida na variação de `localWorkSize` deve-se à concorrência de acesso entre os bancos de memória dentro de uma unidade de computação, causando, assim, um enfileiramento dos acessos em uma fila de instruções *load/store*, situação que deve ser evitada a todo custo a fim de melhorar o desempenho da implementação.

4.2.5 OTIMIZAÇÕES

Este capítulo se concentra na implementação de algumas técnicas de otimização de algoritmos massivamente paralelos, tais técnicas devem ser observadas ao desenvolver algoritmos nessa arquitetura de hardware, uma vez que, o sucesso da implementação depende da correta adequação do algoritmo a essa arquitetura.

Com o objetivo de verificar o impacto dos diversos métodos de otimizações que se aplicam à programação massivamente paralela, foram implementados diversos algoritmos e os desempenhos da rotina Jacobi foram analisados em cada caso, cujos resultados serão apresentados no próximo capítulo.

4.2.5.1 ACESSO COALESCENTE À MEMÓRIA

O conceito de acesso coalescente está relacionado com a forma que as GPUs realizam os acessos à memória. Quando uma porção da memória precisa ser acessada, todas as threads que compõem o *workgroup* realiza o acesso ao banco de memória sequencialmente. Porém, caso seja possível ordenar os dados de modo que o acesso à memória de cada thread seja contíguo, então, todas as threads do *workgroup* irão buscar seus dados em um único acesso, quando isso acontece, o acesso é dito coalescente.

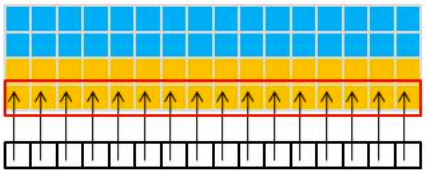


Figura 18: Acesso coalescente à memória

A figura 18 ilustra um acesso coalescente à memória. É importante observar que cada *thread* faz acesso à posição do *array* correspondente ao seu identificador, assim, ao invés do acesso ser feito sequencialmente por cada *thread*, ele é feito em blocos de *threads* ou *wavefronts*, é importante lembrar que caso alguma thread do mesmo bloco esteja desalinhada, todo o acesso será sequencial, portanto, sem o benefício do acesso coalescente.

Como exemplo prático, tem-se um *kernel* simples que recebe um *array* `v`, localizado na memória global, e armazena um valor desse *array* na variável `val`. Tem-se o código com acesso coalescente e outro com acesso não coalescente.

- Acesso coalescente

```
1 __kernel void coalesced(__global float * v) {
2     int i = get_global_id(0);
3     float val = v[i];
4 }
```

- Acesso não coalescente

```
1  __kernel void noncoalesced(__global float * v) {
2
3     int i = get_global_id(0);
4     float val = v[2*i];
5 }
```

Uma observação importante relacionada ao acesso não coalescente do exemplo acima é que ele não acessa a memória de forma contígua, os primeiros acessos são para as regiões: `v[0]`, `v[2]`, `v[4]`, etc.

4.2.5.2 UTILIZAÇÃO DA MEMÓRIA LOCAL

Como explicado no capítulo 3, a GPU possui uma memória de baixa latência localizada no próprio chip, denominada memória local. Apesar da vantagem no acesso rápido à esse tipo de memória, existe uma limitação de seu tamanho devido ao seu alto custo. No *hardware* utilizado presente trabalho esta memória possui 32KB.

Para o melhor entendimento dos benefícios do uso à memória local, pode-se fazer uma analogia no universo das CPUs.

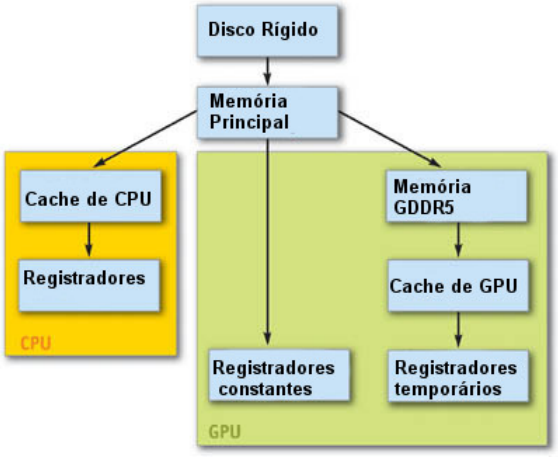


Figura 19: Memória local e CPU *cache*

A figura 19 ilustra a organização dos diversos tipos de memória disponíveis na arquitetura de computadores.

Quando um programa de CPU precisa buscar algum dado que está localizado na memória principal, ele faz um cópia de um bloco de memória que contenha o dado solicitado na memória *cache*. Esse procedimento faz com que caso o programa necessite de um outro dado da memória, ele primeiro verifica se o dado está na *cache*, caso esteja, a busca é otimizada uma vez que a memória cache está localizada no próprio chip, sendo uma memória de baixa latência.

Analogamente ao caso em CPU, a GPU possui uma memória de baixa latência localizada no próprio chip, chamada de memória local. Por ser uma memória no próprio chip, possui alto custo, sendo portante pequena. Da mesma maneira que é feito na memória cache da CPU, pode-se realizar uma cópia dos dados da memória global do dispositivo na memória local, após a manipulação dos dados nessa memória, é realizada uma cópia para a memória global, esse procedimento é feito manualmente pelo programador.

Como explicado anteriormente, para utilizar a memória local, o dado localizado na memória global deve ser copiado para um *array* localizado na memória local, e este deve ser utilizado até o momento em que o resultado estiver pronto para ser enviado para a memória global novamente.

Suponha que se queira realizar a multiplicação de matrizes usando GPU. Seguem os códigos utilizando memória global e utilizando memória local.

- Utilizando memória global

```
1  __kernel void simpleMultiply(__global float* a, __global float* b,
2  __global float* c, int N) {
3  int row = get_global_id(1);
4  int col = get_global_id(0);
5  float sum = 0.0f;
6  for (int i = 0; i < TILE_DIM; i++) {
7    sum += a[row*TILE_DIM+i] * b[i*N+col];
8  }
9    c[row*N+col] = sum;
10 }
```

- Utilizando memória local

```
1  __kernel void coalescedMultiply(double*a, double* b, double*c, int N)
2  {
3  __local float aTile[TILE_DIM][TILE_DIM];
4  __local double bTile[TILE_DIM][TILE_DIM];
5
6   int row = get_global_id(1);
7   int col = get_global_id(0);
8   float sum = 0.0f;
9   for (int k = 0; k < N; k += TILE_DIM) {
10    aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];
11    bTile[threadIdx.y][threadIdx.x] = b[threadIdx.y*N+col];
12    barrier(CLK_LOCAL_MEM_FENCE);
13
14    for (int i = k; i < k+TILE_DIM; i++) {
15      sum += aTile[threadIdx.y][i]* bTile[i][threadIdx.x];
16    }
17  }
18  c[row*N+col] = sum;
19 }
```

É importante observar no algoritmo que utiliza memória local a cópia dos dados da memória global na memória local, a manipulação dos dados utilizando a memória local e a posterior cópia do resultado para a memória global.

4.2.5.3 MAPAMENTO DE MEMÓRIA NO HOST

Em linguagem de programação C99 existe um recurso que permite que um arquivo seja mapeado na memória fazendo com que a manipulação dos dados ali contidos possa ser realizada através de funções de manipulação de memória que são implicitamente replicadas no arquivos em questão. Na linguagem OpenCL, é possível fazer um procedimento semelhante, em que ao invés de um arquivo, usa-se a memória do dispositivo, assim, realizando o mapeamento da memória do dispositivo no *host*, usa-se funções de manipulação de memória no *host*, sendo replicadas implicitamente para a memória do dispositivo, esse procedimento se chama mapeamento de memória.

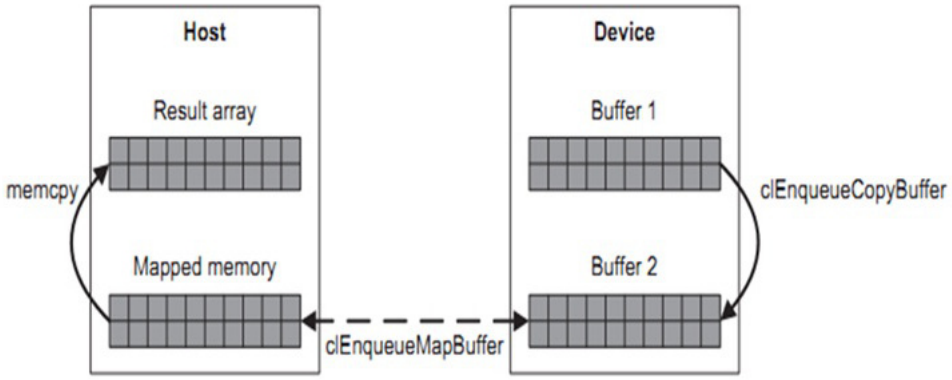


Figura 20: Mapeamento de memória [fonte: Scarpino(2011)]

A figura 20 ilustra o procedimento, são criados dois *buffers* de memória no dispositivo, sendo que o `buffer2` recebe os dados do `buffer1` através do comando `clEnqueueCopyBuffer`, então, no *host*, um *array* é alocado e mapeado em `buffer2`, após isso, o comando de memória `memcpy` que é responsável por copiar uma região da memória em outra é utilizado para copiar os dados da região mapeada para um *array* resultado, implicitamente, o mapeamento copia a região de memória do dispositivo para o *array* resultado. Assim, ao final do processo, o *array* resultado estará com os mesmos dados do `buffer2` contido no dispositivo, realizando cópia explícita.

4.2.5.4 OTIMIZAÇÃO AUTOMÁTICA DO COMPILADOR

Em ciência da computação, o termo otimização de compilação é utilizado para designar um processo de compilação que procura alterar alguns atributos do programa executável, geralmente, o tempo de execução do programa. Atualmente, é um assunto de muito estudo, uma vez que a otimização dos programas pode levar a um consumo reduzido de recursos de hardware fazendo com que o tempo de bateria em equipamentos móveis seja maior.

Segundo a empresa fabricante AMD, quando a otimização automática do compilador está ativa, o compilador tenta aplicar uma série de otimizações ao código para diminuir o tempo de execução ou o tamanho do programa binário.

Entre as otimizações implementadas, estão a vetorização dos tipos de dados, isto é, os dados de *arrays* e matrizes são compactados em tipos de dados vetoriais (`int2`, `int3`, `int4`, `float2`, `float3`, `float4`, `double2`, `double3` ou `double4`), assim, cada operação é realizada em grupos de dados e não em dados individuais, favorecendo o tempo de processamento.

4.2.6 ALGORITMOS IMPLEMENTADOS

No presente trabalho foram desenvolvidos duas implementações do método de Jacobi.

A primeira implementação, resolve o sistema de equações lineares utilizando paralelização trivial (noline), ou seja, cada *thread* é responsável pela solução de uma variável `x`. Neste caso, observando a figura 21, cada *thread* do algoritmo calcula cada solução de `x` independentemente, ou seja, cada *thread* realiza todo o processo de multiplicação de cada elemento da matriz `A` com cada elemento do vetor `x` e atualiza a possível solução `x` até que seu valor esteja abaixo do valor tolerância, atingindo esse limite, a iteração cessa.

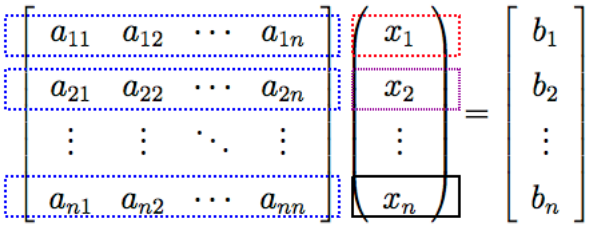


Figura 21: Diagrama de paralelização trivial (noline)

Por outro lado, a segunda implementação, realiza a paralelização por linha, isto é, cada bloco de *thread* é responsável por uma linha da matriz `A`. Neste caso, observando a figura 22, o algoritmo percorre cada linha da matriz `A` (identificada com a linha constante) e realiza a multiplicação com o vetor `x`, então o possível resultado de `x` é atualizado a cada iteração até que seu valor esteja abaixo do valor tolerância, atingindo esse limite, a iteração cessa.

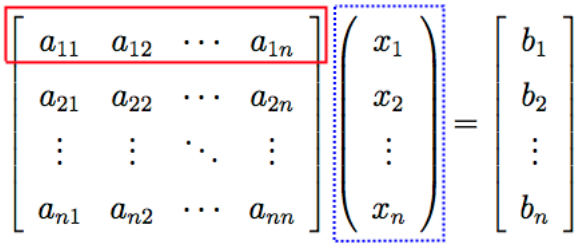


Figura 22: Diagrama de paralelização por linha (line)

5 RESULTADOS

Neste capítulo são apresentados os resultados obtidos com o trabalho desenvolvido nos capítulos anteriores.

Os resultados obtidos com o método dos elementos de contorno são diretamente relacionados com o grau de discretização dos elementos que compõe o problema, isto é, quanto maior o nível de discretização, mais próximos dos valores analíticos serão os resultados numéricos.

A título de verificação da conformidade com os resultados esperados, é apresentada a seguir um exemplo de saída do programa para uma simulação do problema potencial com discretização de 4096 nós de contorno, executado em GPU utilizando 100 pontos internos, mesclando as saídas de potencial e gradiente de potencial.

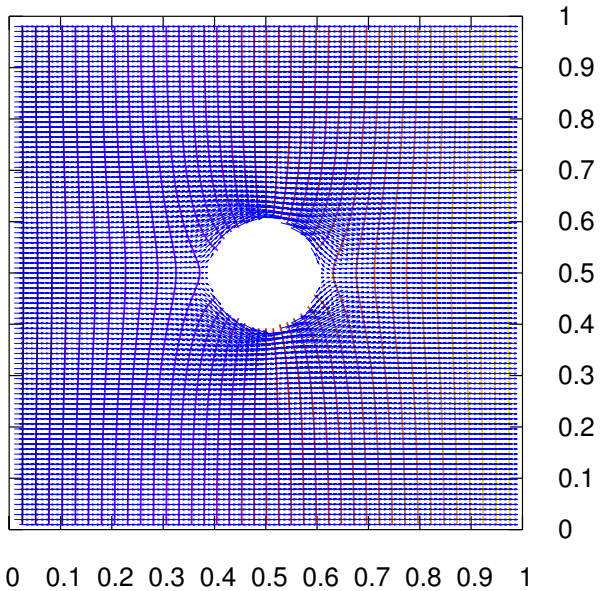


Figura 23: Campo de potenciais e gradientes de potencial.

Desmembrando os resultados do gráfico 23, é possível verificar as duas saídas da implementação. Primeiramente na figura 24, apresentamos o gráfico de potenciais para o caso de 4096 nós de contorno, utilizando 100 pontos internos, executado em GPU.

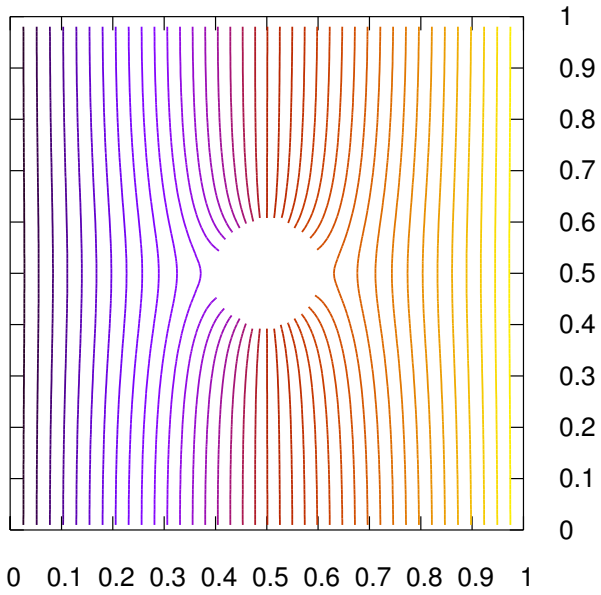


Figura 24: Campo de potenciais.

Para o caso do gradiente da função potencial, o número de pontos internos foi reduzido para 50, a fim de obter uma maior clareza das derivadas da função potencial, porém os outros parâmetros do problema foram mantidos.

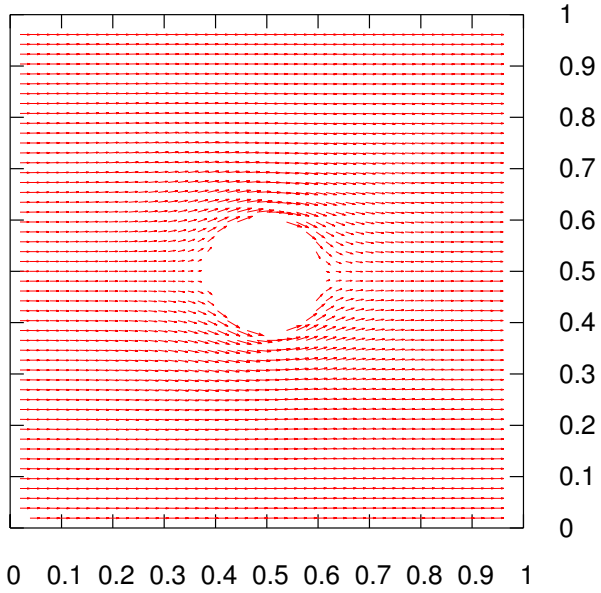


Figura 25: Campo de gradientes de potencial.

A figura 24 mostra o comportamento do fluxo potencial para o problema abordado.

Com o objetivo de analisar o desempenho do algoritmo proposto, foram realizados diversos níveis de discretização das bordas, avaliando os tempos de execução da rotina de Jacobi, sendo que para cada valor apresentado foi realizada uma média aritmética de 10 execuções a fim de minimizar os erros estatístícos. Além disso, foi utilizado o conceito

de velocidade, definindo, assim, uma maneira de verificar o custo computacional de cada ponto do problema (11). Esse parâmetro de desempenho é definido como:

$$Velocidade = \frac{nnos \times nnos}{tempo}. \tag{5.1}$$

onde *nnos* é quantidade de nós de discretização e tempo é a quantidade de *tempo* de duração da execução do algoritmo, em segundos.

Após diferentes implementações com diferentes níveis de otimização, e utilizando como base a variável Ganho (definida como Ganho, onde $velocidade_{serial}$ é a média das velocidades encontradas no algoritmo serial quando o hardware estava completamente saturado, ou seja, no estado do platô) para avaliar o desempenho dos mesmos. A figura 26 ilustra como foi obtido o valor velocidade serial a fim de obter a média e definir a variável ganho.

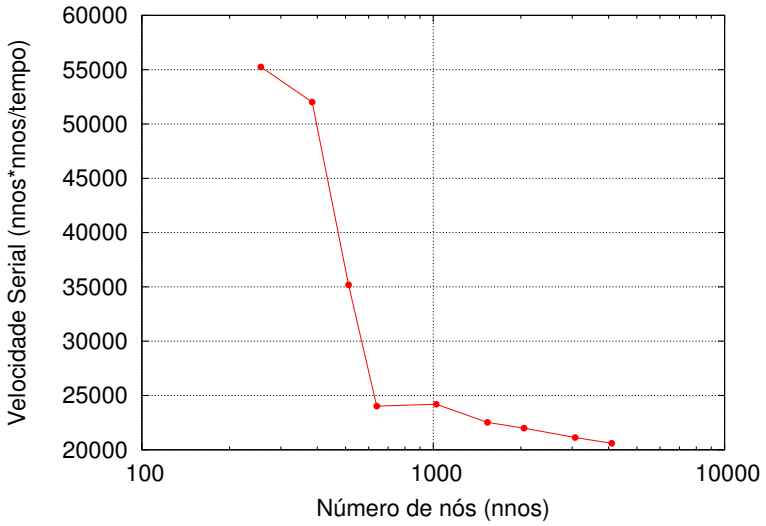


Figura 26: Velocidade do algoritmo serial

A média dos valores da velocidade serial foi obtida tomando a média aritmética dos pontos do platô (linha aproximadamente horizontal do gráfico 26), obtendo o valor 2.31837825E+04 ($\frac{nnnos \times nnos}{tempo}$). Assim, o ganho é definido como:

$$Ganho = \frac{velocidade}{velocidade_{serial}}. \tag{5.2}$$

Além disso, foram utilizadas as siglas CO para o algoritmo com otimização do compilador e SO para o algoritmo sem a otimização do compilador, assim, foram obtidos os seguintes resultados.

5.1 PARALELIZAÇÃO TRIVIAL

Foram executadas médias aritméticas utilizando 10 execuções para cada ponto considerado, obtendo os dados apresentados na tabela 1.

Tabela 1: Ganho do algoritmo de Jacobi (em relação ao serial) pelo número de nós (paralelização trivial)

N. nós	GPUSO(Ganho)	8coreSO(Ganho)	GPUCO(Ganho)	8coreCO(Ganho)
256	0.91	1.38	0.80	1.69
384	1.58	1.34	1.68	3.14
512	2.06	1.88	2.13	6.58
640	2.71	1.61	2.97	6.67
1024	1.78	1.93	1.89	5.09
1536	1.89	1.91	1.98	3.99
2048	1.08	1.86	1.09	3.97
3072	2.80	1.76	2.94	3.86

O resultado da figura 27 mostra que o tempo de processamento do algoritmo em CPU (ganho) com otimização do compilador e utilizando 8 núcleos de processamento é quase 4 vezes menor que o algoritmo serial, em contra partida, o tempo de execução do mesmo algoritmo sem a otimização do compilador é apenas duas vezes menor que do algoritmo serial. Por outro lado, os algoritmos em GPU são pouco afetados pela otimização automática do compilador.

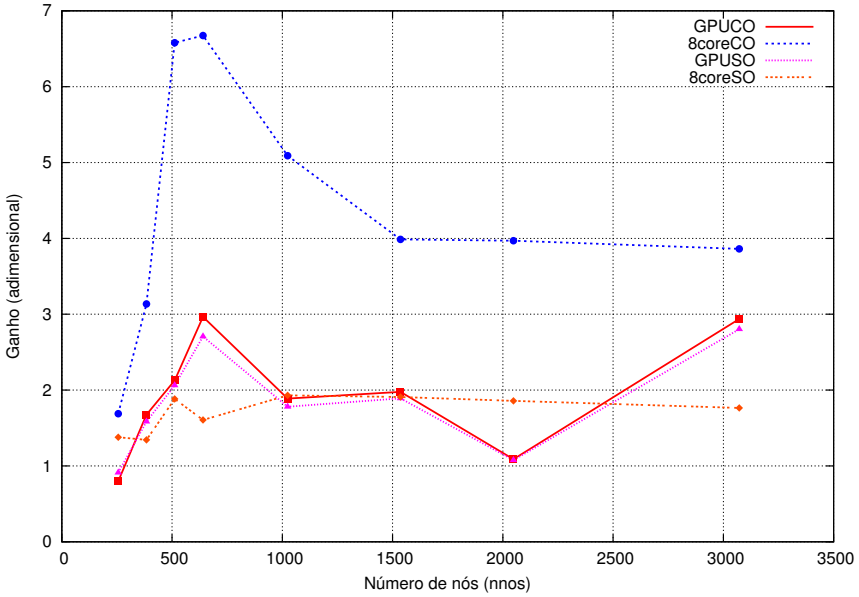


Figura 27: Paralelização trivial com mapeamento.

Esse resultado mostra a importância da utilização das variáveis de otimização automática do compilador quando se utiliza códigos *multithreads* em CPU e que o mesmo efeito não acontece em algoritmos executados em GPU.

5.2 CÓDIGO LINHA COM X E A NA MEMORIA GLOBAL COM MAPEAMENTO

Foram executadas médias aritméticas utilizando 10 execuções para cada ponto considerado, obtendo os dados apresentados na tabela 2.

Tabela 2: Ganho do algoritmo de Jacobi (em relação ao serial) pelo número de nós (X e A na memória global - mapeada)

N. nós	GPUSO(Ganho)	8coreSO(Ganho)	GPUCO(Ganho)	8coreCO(Ganho)
256	1.36	0.46	1.14	1.86
384	2.52	0.59	2.69	3.62
512	3.74	0.61	4.06	3.78
640	4.69	0.67	5.23	4.01
1024	4.51	0.57	4.78	3.89
1536	3.86	0.60	4.94	3.92
2048	4.21	0.57	5.46	3.86

O resultado da figura 28 mostra que quando se utiliza memória mapeada no *host* e quando tanto `x` quanto `A` estão na memória global, o algoritmo em GPU com otimização é quase 5.5 vezes mais rápido do que o algoritmo serial. Esse ganho com as variáveis na memória global é justificável pelo fato do acesso coalescente à memória fazer com que o *throuput* seja muito maior do que se as variáveis estivessem na memória local do dispositivo, isto é, a quantidade de dados lidos ou escritos na memória é muito maior no mesmo período de tempo.

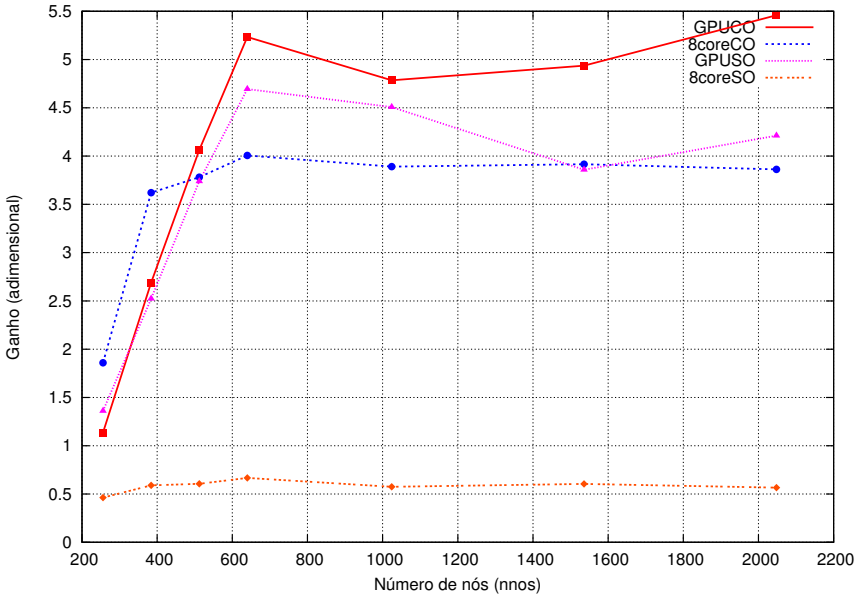


Figura 28: Código linha (x e A na mem. global) com mapeamento.

Para o algoritmo em CPU utilizando otimização automática, não houve grande diferença entre o algoritmo utilizando paralelização trivial e paralelização linha a linha do sistema de equações lineares. Porém, observa-se que o mesmo algoritmo sem otimização leva o dobro de tempo para ser executado, em relação ao algoritmo serial.

5.3 CÓDIGO LINHA COM X E A NA MEMORIA GLOBAL SEM MAPEAMENTO

Foram executadas médias aritméticas utilizando 10 execuções para cada ponto considerado, obtendo os dados apresentados na tabela 3.

Tabela 3: Ganho do algoritmo de Jacobi (em relação ao serial) pelo número de nós (X e A na memória global - não mapeada)

N. nós	GPUSO(Ganho)	8coreSO(Ganho)	GPUCO(Ganho)	8coreCO(Ganho)
256	1.19	0.45	1.23	1.74
384	2.20	0.59	2.36	3.60
512	3.34	0.60	3.57	3.94
640	4.25	0.66	4.71	3.99
1024	4.50	0.56	4.75	3.88
1536	3.77	0.60	4.73	3.88
2048	3.80	0.57	4.79	3.78

A figura 29 mostra que quando se utiliza memória não mapeada no *host* e quando tanto `x` quanto `A` estão na memória global o algoritmo em GPU com otimização é quase 5 vezes mais rápido que o algoritmo serial e que a velocidade do algoritmo em GPU sem otimização fica extremamente próximo do algoritmo de CPU com otimização. Além disso, o algoritmo em CPU sem otimização continua tendo um maior tempo de processamento em relação ao algoritmo serial.

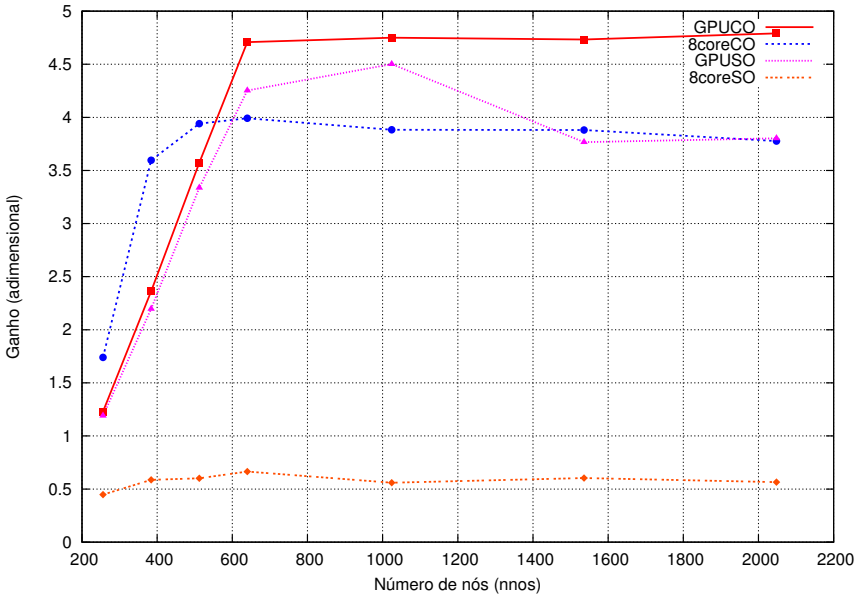


Figura 29: Código linha (x e A na mem. global) sem mapeamento.

Para o caso do algoritmo com paralelização das linhas do sistema de equações lineares houve uma pequena melhora de desempenho ao utilizar o mapeamento de memória no *host*.

5.4 CÓDIGO LINHA COM X E A NA MEMÓRIA LOCAL COM MAPEAMENTO

Foram executadas médias aritméticas utilizando 10 execuções para cada ponto considerado, obtendo os dados apresentados na tabela 4.

Tabela 4: Ganho do algoritmo de Jacobi (em relação ao serial) pelo número de nós (X e A na memória local - mapeada)

N. nós	GPUSO(Ganho)	8coreSO(Ganho)	GPUCO(Ganho)	8coreCO(Ganho)
256	1.27	0.31	1.37	2.11
384	1.80	0.37	2.41	2.66
512	2.67	0.38	3.14	2.68
640	2.55	0.39	3.27	3.00
1024	2.42	0.37	3.08	2.93
1536	1.48	0.37	2.04	2.82
2048	1.58	0.36	2.13	2.68

A figura 30 mostra que quando `A` e `x` são mantidos na memória local o algoritmo em CPU com otimização do compilador obtém o melhor desempenho em relação ao algoritmo serial, sendo aproximadamente 2.7 vezes mais rápido do que o algoritmo serial, apresentando um desempenho inferior ao obtido com as mesmas variáveis na memória global. O algoritmo em CPU sem otimização continuou apresentando um desempenho inferior ao algoritmo serial.

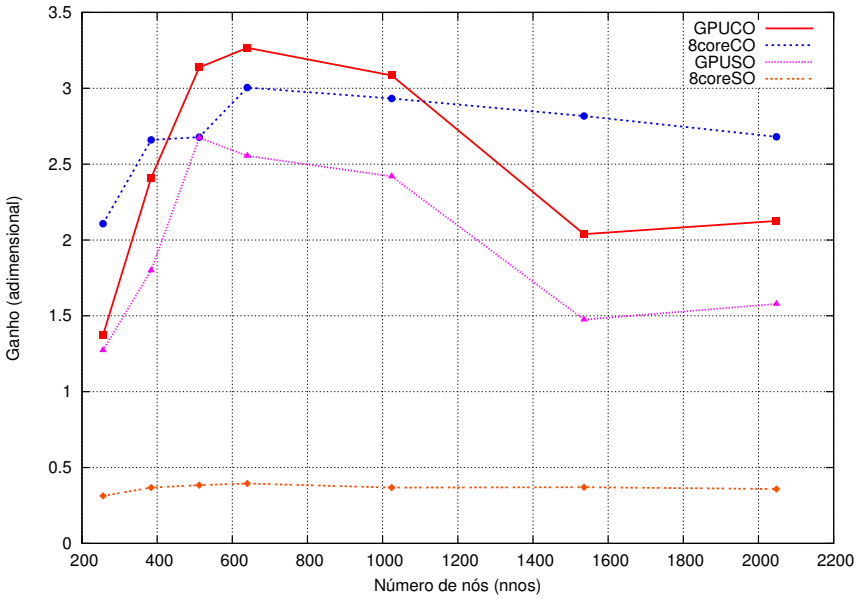


Figura 30: Código linha (x e A na mem. local) com mapeamento.

O algoritmo em GPU apresentou uma subtancial degradação no desempenho ocasionado principalmente pelo baixo *throughput* oferecido pela memória local que apesar de possuir baixa latência, é pequena.

5.5 CÓDIGO LINHA COM X E A NA MEMÓRIA LOCAL SEM MAPEAMENTO

Foram executadas médias aritméticas utilizando 10 execuções para cada ponto considerado, obtendo os dados apresentados na tabela 5.

Tabela 5: Ganho do algoritmo de Jacobi (em relação ao serial) pelo número de nós (X e A na memória local - não mapeada)

N. nós	GPUSO(Ganho)	8coreSO(Ganho)	GPUCO(Ganho)	8coreCO(Ganho)
256	1.11	0.30	1.18	2.12
384	1.90	0.36	2.04	2.65
512	2.45	0.38	2.80	2.60
640	2.48	0.39	3.17	3.03
1024	2.34	0.37	2.89	2.93
1536	1.52	0.37	2.01	2.84
2048	1.55	0.36	2.08	2.68

A figura 31 mostra que o melhor desempenho continuou sendo do algoritmo em CPU com otimização, observa-se, também, que a utilização de mapeamento de memória no *host* não apresentou uma variação expressiva no desempenho.

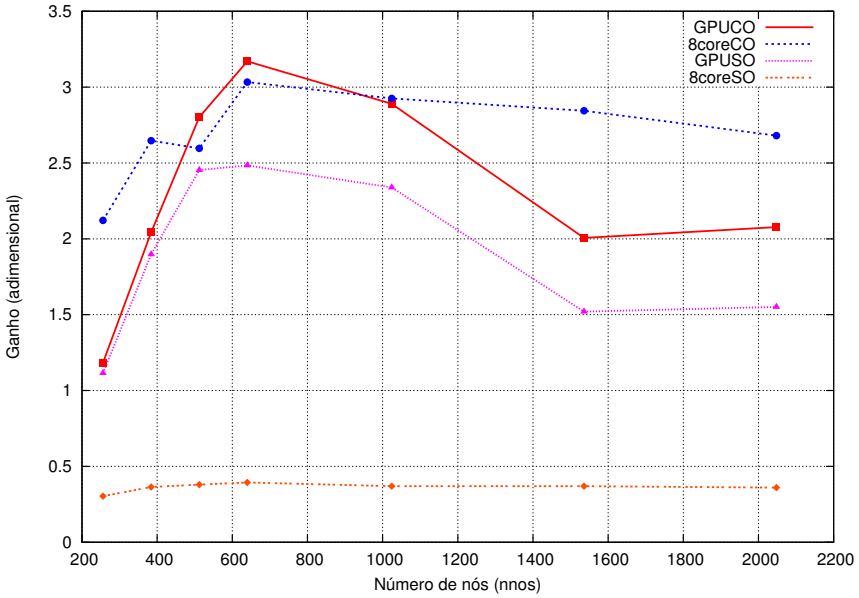


Figura 31: Código linha (x e A na mem. local) sem mapeamento.

5.6 MELHOR GANHO DE CADA ARQUITETURA

Foram executadas médias aritméticas utilizando 10 execuções para cada ponto considerado, obtendo os dados apresentados na tabela 6.

Tabela 6: Ganho do algoritmo de Jacobi (em relação ao serial) pelo número de nós (melhor ganho)

N. nós	GPUSO(Ganho)	8coreSO(Ganho)
256	1.14	1.69
384	2.69	3.14
512	4.06	6.58
640	5.23	6.67
1024	4.78	5.09
1536	4.94	3.99
2048	5.46	3.97

A figura 32 mostra o melhor resultado de cada uma das arquiteturas obtido com toda a análise de otimizações realizadas neste trabalho. Em ambos os algoritmos, o melhor resultado foi aquele com otimização de compilação. No caso de GPU esse fator não interfere no resultado, porém, no caso CPU é um fator fundamental de desempenho. Além disso, para o caso GPU, o melhor resultado foi obtido mantendo a variável `X` na memória global e utilizando mapeamento no *host* (no gráfico, identificado como XGM), enquanto o algoritmo em CPU que ofereceu o melhor desempenho foi aquele com `A` e `X` na memória local e memória não mapeada no *host* (no gráfico, identificado como AXLU).

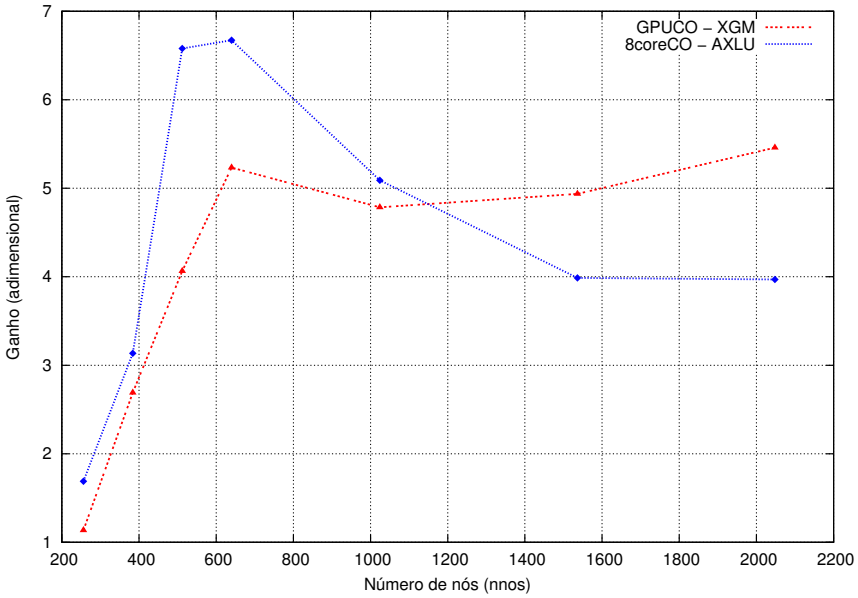


Figura 32: Melhor ganho.

Para o caso CPU observa-se que houve um ápice de desempenho com número de nós próximo a 512, esse comportamento é justificável pelo modo com que a CPU realiza as buscas dos dados na memória. Ao realizar uma busca na memória, o processador busca além do dado solicitado, os dados próximos a ele realizando o armazenamento na memória cache (memória rápida de baixa latência). Sendo assim, no caso apresentado na figura 32, a CPU consegue armazenar uma grande parte dos dados a serem processados, fazendo com que um ápice no processamento deste caso isolado seja identificação, fato que não acontece quando o número de nós aumenta.

É importante observar que o algoritmo que utiliza paralelização trivial obteve o melhor desempenho, em relação ao algoritmo serial, quando executado em CPU, ao passo que o algoritmo com paralelização das linhas da matriz obteve o melhor resultado, em relação ao algoritmo serial, quando executado em GPU.

6 CONCLUSÕES

O objetivo deste trabalho foi o desenvolvimento de uma biblioteca de alto desempenho para a resolução da Equação de Laplace bidimensional com condições de contorno mistas, utilizando o método dos elementos de contorno, como exemplo de utilização da biblioteca, foi abordado o problema de um fluido potencial que escoa em torno de um cilindro circular maciço. O problema utilizou o método dos elementos de contorno baseado na referência (17), porém, o algoritmo foi melhorado e otimizado. Dentre as rotinas desenvolvidas, foi criado um conjunto de rotinas de pré-processamento a fim de determinar as configurações dos pontos geométricos dos contornos e atribuir suas condições de contorno, além de realizar a geração de grid dos pontos internos ao contorno de maneira automática. De modo a simplificar o cálculo, o campo de gradiente de potencial foi determinado utilizando-se o método de diferenças finitas na derivação da função potencial.

Em um primeiro momento, um código serial em linguagem MATLAB foi desenvolvido, cujos resultados obtidos foram validados utilizando um programa funcional, desenvolvido na referência (17) na linguagem FORTRAN. Após a fase de validação dos modelos numéricos, o código serial foi portado para a linguagem de programação C e, posteriormente, para a linguagem OpenCL, linguagem essa semelhante à linguagem C99, ocorrendo assim uma nova validação dos resultados.

Foram implementadas diversas técnicas de otimização a fim de avaliar seu comportamento na CPU e GPU. O acesso coalescente à memória mostrou-se de grande impacto no desempenho do algoritmo e o mapeamento do memória no *host* forneceu um ganho de dempenho em relação ao algoritmo sem mapeamento de memória. Outro fator avaliado foi a utilização da otimização automática do compilador, neste caso, seu uso utilizando CPU mostrou ser muito vantajoso, pois os recursos multithreads se tornam melhor aproveitados enquanto que na GPU esse parâmetro não apresenta ganho de desempenho.

Os resultados mostram que a utilização da linguagem OpenCL com os algoritmos propostos oferecem um ganho em relação ao algoritmo serial de até 5.5 vezes. Porém,

é importante observar que um algoritmo que forneça um ganho na GPU nem sempre oferecerá um ganho quando executado em CPU, esse fato deve-se à própria arquitetura do *hardware*, isto é, as técnicas de otimização a serem utilizadas devem levar em consideração em que arquitetura o código será executado, este é um importante resultado do presente trabalho. A arquitura da GPU é inerentemente paralela e, como tal, minimizar as trocas de memória e manter o *hardware* saturado o maior tempo possível levará a um ganho elevado, pois muitos dados serão tratados simultaneamente. Por outro lado, um código que sature uma CPU executará uma série de alocações e desalocações de recursos, isto é, uma série de trocas de contexto, sendo menos eficiente que o código executado em GPU, implicando diretamente na técnica de otimização a ser utilizada.

Além disso, neste trabalho, foi realizado todo o desenvolvimento matemático do método dos elementos de contorno necessário para a implementação de modelos baseados em problemas potenciais, sendo possível, por exemplo, a modelagem do problema de um escoamento potencial em torno de um cilindro.

Segundo a referência (28), o agendador de tarefas da GPU da fabricante ATI é eficiente na paralelização de grande volume de dados, onde a latência na cópia de dados entre as memórias é mascarada com o uso de várias frentes de onda, isto é, caso uma operação em um dado *stream processor* esteja aguardando uma informação vinda da memória, ela é substituída por outra operação que irá manter o *stream processor* ativo até que a informação chegue. Porém, para que a latência seja mascarada, é necessário que o *workitem* possua uma grande quantidade de operações lógico-aritméticas quando comparada com as operações de acesso a memória (*load/store*).

Apesar da documentação da fabricante afirmar que o *hardware* está preparado para operar um grande volume de dados, a implementação do OpenCL para a fabricante ATI possui uma séria limitação na quantidade de memória alocada pela rotina `clCreateBuffer`. Esse fator limitante fez com que as simulações realizadas neste trabalho não saturassem o dispositivo da maneira que era esperada devido a impossibilidade de simular mais de 4096 pontos de grid, pois acima deste valor, não é possível alocar os *arrays* da matriz de coeficientes do sistema (nnos×nnos).

O *hardware* utilizado possui 1 GB de memória GDDR3 dedicada. Porém, os *drivers* do OpenCL para a ATI permitem a utilização de apenas 50% dessa quantidade, das quais apenas $\frac{1}{4}$ pode ser alocada por vez. Sendo assim, tem-se um limite de 128 MB para cada alocação, tornando-se um grande fator limitante na utilização da atual implementação OpenCL.

Uma possível solução para o problema de alocação de memória identificado no trabalho seria a alocação de diversos vetores múltiplos de 128MB (limite de alocação da implementação OpenCL 1.2) no dispositivo de forma contígua e, após a alocação destes *arrays*, realizar o acesso através de um ponteiro que aponta para o início do bloco da memória alocada, assim como na implementação C99, seria possível acessar todo o grande bloco de memória do dispositivo, superando, assim, a limitação citada.

Outra possibilidade seria explorar mais a fundo o processo de alocação de memória em linguagem de baixo nível utilizando o manual de referência da linguagem intermediária da fabricante AMD ATI (29). Apesar dessa linguagem intermediária ser muito próxima à linguagem Assembly, ela permite o acesso direto ao *hardware*, sendo possível otimizações impraticáveis através da linguagem OpenCL.

Existe também na comunidade científica o início de uma tendência de se aplicar nas placas de vídeo os conceitos de virtualização que já estão bem consolidados no universo da computação (30). Essa tendência pode ser exemplificada pela referência (31), que propõe um *framework* capaz de tratar um sistema de múltiplas GPUs como se fosse uma única, de forma transparente ao usuário. Assim, um tratamento é realizado a nível de tempo de execução, tornando possível alocar toda a memória dos dispositivos como se fosse uma única memória virtual, além de utilizar toda a capacidade de processamento do conjunto.

Para a continuidade do trabalho, é importante observar outras técnicas de otimização, como a vetorização explícita. A vetorização é uma técnica de processamento inerentemente paralela e remete aos antigos processadores gráficos que faziam essencialmente esse tipo de operação, através da vetorização é possível executar uma operação em um grupo de dados em um único ciclo de *clock*. Além disso, como sugere o gráfico da figura 14, a rotina de construção do sistema de equações lineares deve ser paralelizada a fim de obter um menor tempo de processamento global da biblioteca. É importante observar, também, que existem outras arquiteturas que suportam a linguagem OpenCL como o Intel Phi e as placas gráficas da NVIDIA, sendo assim, avaliar o comportamento do algoritmo nessas arquiteturas alternativas fornecerão bases para a escolha do *hardware* que melhor atenderá ao problema.

Referências

1 CASTOLDI A., E. GATTI, and P. REHAK. Three-dimensional analytical solution of the laplace equation suitable for semiconductor detector design. *IEEE Transactions on Nuclear Science*, 43, 1996.

2 ACHARYYA A., P. BAISAKHI, and J. P. BANERJEE. Temperature distribution inside semi-infinite heat sinks for impatt sources. *International Journal of Engineering Science and Technology*, 2(10), 2010.

3 B. GENG. Flow field's laplace equation and analysis. *International Conference on Electronics and Optoelectronics*, 2, 2011.

4 BUTKOV E. I. *Mathematical Physics*, volume 1. Addison-Wesley, 1968.

5 ZIENKIEWICZ O.C. and R.L. TAYLOR. *The Finite Element Method*, volume 1. Butterworth-Heinemann, 2000.

6 BARCELOS M. and MAUTE K. Aeroelastic design optimization for laminar and turbulent flows. *Computer Methods in Applied Mechanics and Engineering*, 197, 2008.

7 CAUSON D. M., MINGHAM C. G., and L. QIAN. *Introductory Finite Volume Methods for PDEs*, volume 1. Ventus Publishing ApS, 2011.

8 BIJELONJA I., DEMIRDZIC I., and MUZAFERIJA S. A finite volume method for incompressible linear elasticity. *Computer Methods in Applied Mechanics and Engineering*, 195, 2006.

9 GONZA P., CABALEIRO J. C., and PENA T. F. Parallel iterative solvers involving fast wavelet transforms for the solution of bem systems. *Advances in Engineering Software*, 33, 2002.

10 Top 500 supercomputer sites. disponivel em http://www.top500.org, 05/03/2013.

11 BARROS A., GARCIA E., and MORGADO R. Simulation of stochastic processes using graphics hardware. *Computer Physics Communications*, 2010.

12 BLOCK B., VIRNAU P., and PREIS T. Multi-gpu accelerated multi-spin monte carlo simulations of the 2d ising model. *Computer Physics Communications*, 181, 2010.

13 MOLNAR F., SZAKALY T., MESZAROS R., and LAGZI I. Air pollution modelling using a graphics processing unit with cuda. *Computer Physics Communications*, 181, 2010.

14 ANDERSON J. A., LORENZ C. D., and TRAVESSET A. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227, 2008.

15 HWU W. and KIRK D. *Programming Massively Parallel Processors*. Elsevier, 2010.

16 LIU Y. *Fast Multipole Boundary Element Method Theory and Applications in Engineering*. Cambridge University Press, 2009.

17 KATSIKADELIS J. T. *Boundary Elements Theory and applications*. ELSEVIER, 2002.

18 ISO. Iso c standard 1999. Technical report. ISO/IEC 9899:1999 draft.

19 SCARPINO M. *OpenCL in Action*, volume 1. Addison-Wesley Professional, 2011.

20 The opencl specification, 2012. Version 1.2.

21 IEEE. *IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. 1995.

22 DAGUM L. and MENON R. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

23 GROPP W., EWING L., and ANTHONY S. *Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface (Scientific and Engineering Computation)*. The MIT Press, second edition edition, 1999.

24 YOU S. and XIE X. A synchronous jacobi iteration parallel algorithm for solving liner system based on scilab. *Anti-Counterfeiting Security and Identification in Communication (ASID), 2010 International Conference on*, 2010.

25 CAPUZZO-DOLCETTA R., SPERA M., and PUNZO D. A fully parallel, high precision, n-body code running on hybrid computing platforms. *Journal of Computational Physics*, 236, 2013.

26 NAKASATO N. A fast gemm implementation on the cypress gpu. *ACM SIGMETRICS Performance Evaluation Review*, 38, 2011.

27 OBRECHT C., KUZNIK F., TOURANCHEAU B., and ROUX J. Multi-gpu implementation of a hybrid thermal lattice boltzmann solver using the thelma framework. *Computers and Fluids*, 2012.

28 *AMD Accelerated Parallel Processing OpenCL*. Advanced Micro Devices, 2012.

29 *Compute Abstraction Layer (CAL) Technology Intermediate Language (IL)*. Advanced Micro Devices, 2010.

30 GRINBERG S. and WEISS S. Architectural virtualization extensions: A systems perspective. *Computer Science Review*, 6, 2011.

31 KIM J., KIM H., LEE J. H., and LEE J. Achieving a single compute device image in opencl for multiple gpus. *SIGPLAN Notices*, 46, 2011.