Pontifícia Universidade Católica
do Rio de Janeiro

Andre Luis Cavalcanti Bueno

Resolução de sistemas de equações lineares de grande porte em clusters multi-GPU utilizando o método do gradiente conjugado em OpenCL™

Dissertação de Mestrado

Dissertação apresentada ao Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC–Rio como requisito parcial para obtenção do grau de Mestre em Informática.

Orientador : Profª. Noemi de La Rocque Rodriguez
Co–Orientador: Profª. Elisa Dominguez Sotelino

Rio de Janeiro
Março de 2013

Pontifícia Universidade Católica
do Rio de Janeiro

Andre Luis Cavalcanti Bueno

Resolução de sistemas de equações lineares de grande porte em clusters multi-GPU utilizando o método do gradiente conjugado em OpenCL™

Dissertação apresentada ao Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC–Rio como requisito parcial para obtenção do grau de Mestre em Informática. Aprovada pela Comissão Examinadora abaixo assinada.

Profª. Noemi de La Rocque Rodriguez
Orientador
Departamento de Informática — PUC–Rio

Profª. Elisa Dominguez Sotelino
Co–Orientador
Departamento de Engenharia Civil — PUC–Rio

Profª. Luiz Fernando Martha
Departamento de Engenharia Civil — PUC-Rio

Prof. Renato Fontoura de Gusmão Cerqueira
Departamento de Informática — PUC-Rio

Prof. José Eugenio Leal
Coordenador Setorial do Centro Técnico Científico — PUC-Rio

Rio de Janeiro, 25 de Março de 2013

Andre Luis Cavalcanti Bueno

Graduou-se em Engenharia de Computação pela PUC-Rio em 2010. Em 2007 fez iniciação científica em métodos de resolução de equações diferenciais. De 2008 a 2011 trabalhou no Laboratório de Inteligência Computacional Aplicada (ICA) pertencente ao departamento de Engenharia elétrica da PUC-Rio. Em 2011 iniciou o mestrado ficando vinculado ao Laboratório Tecgraf na PUC-Rio.

Ficha Catalográfica

Bueno, Andre Luis Cavalcanti

Resolução de sistemas de equações lineares de grande porte em clusters multi-GPU utilizando o método do gradiente conjugado em OpenCL™ / Andre Luis Cavalcanti Bueno; orientador: Noemi de La Rocque Rodriguez; co–orientador: Elisa Dominguez Sotelino. — 2013.

64 f. : il. (color); 30 cm

1. Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2013.

Inclui bibliografia.

1. Informática – Teses. 2. GPGPU. 3. Computação de alto desempenho. 4. Método do gradiente conjugado. 5. Multi-GPU. 6. OpenCL. I. Rodriguez, Noemi de La Rocque. II. Sotelino, Elisa Dominguez. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

Agradecimentos

Às minhas orientadoras Professoras Noemi Rodriguez e Elisa Sotelino pelo aprendizado que pude adquirir em todas as nossas reuniões.

Ao Tecgraf (em especial ao professor Luiz Fernando Martha pela confiança), CAPES e PUC–Rio, pelos auxílios concedidos para que esse trabalho fosse plenamente realizado.

À minha mãe pelo incentivo sem igual.

A todos os colegas, professores e funcionários do Departamento de Informática da PUC–Rio, por me acompanharem nessa jornada.

Resumo

Bueno, Andre Luis Cavalcanti; Rodriguez, Noemi de La Rocque; Sotelino, Elisa Dominguez. **Resolução de sistemas de equações lineares de grande porte em clusters multi-GPU utilizando o método do gradiente conjugado em OpenCL™**. Rio de Janeiro, 2013. 64p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Sistemas de equações lineares esparsos e de grande porte aparecem como resultado da modelagem de vários problemas nas engenharias. Dada sua importância, muitos trabalhos estudam métodos para a resolução desses sistemas. Esta dissertação explora o potencial computacional de mútiplas GPUs, utilizando a tecnologia OpenCL, com a finalidade de resolver sistemas de equações lineares de grande porte. Na metodologia proposta, o método do gradiente conjugado é subdivido em kernels que são resolvidos por múltiplas GPUs. Para tal, se fez necessário compreender como a arquitetura das GPUs se relaciona com a tecnologia OpenCL a fim de obter um melhor desempenho.

Palavras–chave

GPGPU; Computação de alto desempenho; Método do gradiente conjugado; Multi-GPU; OpenCL.

Abstract

Bueno, Andre Luis Cavalcanti; Rodriguez, Noemi de La Rocque (Advisor); Sotelino, Elisa Dominguez (Co-Advisor). **Solving large systems of linear equations on multi-GPU clusters using the conjugate gradient method in OpenCL™**. Rio de Janeiro, 2013. 64p. MSc. Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The process of modeling problems in the engineering fields tends to produce substantiously large systems of sparse linear equations. Extensive research has been done to devise methods to solve these systems. This thesis explores the computational potential of multiple GPUs, through the use of the OpenCL tecnology, aiming to tackle the solution of large systems of sparse linear equations. In the proposed methodology, the conjugate gradient method is subdivided into kernels, which are delegated to multiple GPUs. In order to achieve an efficient method, it was necessary to understand how the GPUs' architecture communicates with OpenCL.

Keywords

GPGPU; High performance computing; Conjugate gradient method; Multi-GPU; OpenCL.

Sumário

Lista de figuras

Lista de tabelas

1
Introdução

Durante os últimos anos, os computadores heterogêneos (compostos de CPUs e GPUs) têm revolucionado a computação. Ao dividir a carga de trabalho entre processadores mais adequados pode-se alcançar um tremendo ganho de desempenho. Grande parte dessa revolução tem sido impulsionada pelo surgimento de processadores de múltiplos núcleos, como as GPUs, algumas delas podem executar mais de um trilhão de operações de pontos flutuante por segundo (teraflops). As GPUs foram criadas para processar belas imagens, porém, também podem ser usadas como ferramenta na área de computação de alto desempenho para aplicações na área científica. Sistemas de computação são uma combinação simbiótica de hardware e software. O hardware não é útil sem um bom modelo de programação. O sucesso das CPUs está atrelado ao sucesso de seus modelos de programação, exemplificado pela linguagem C e seus sucessores. C abstrai de forma muito boa um computador sequencial. Para explorar plenamente computadores heterogêneos é preciso portanto, que o modelo de programação utilizado, tenha também uma boa qualidade de abstração. Essa dissertação vai um pouco além da resolução de sistemas. Busca também uma boa forma de estruturação de uma aplicação OpenCL genérica em um ambiente heterogêneo.

1.1
Motivação

Sistemas de equações lineares esparsos e de grande porte (entende-se por grande porte matrizes de ordem maior ou igual a 10^5) aparecem como resultado da modelagem de vários problemas nas engenharias. Dada sua importância, muitos trabalhos estudam métodos para a resolução dos mesmos [1,2,3,4,5].

Considere um sistema linear,

$$Ax = B \tag{1-1}$$

onde A é uma matriz com elementos reais, de ordem n, quadrada e positiva definida. Uma maneira de resolver (1-1) seria utilizando por exemplo a decomposição QR e retro-substituição ou ainda a decomposição LU da matriz

A [6]. O custo computacional da decomposição de A em LU, por exemplo, seria $O(n^3)$. Isso é razoável quando A é pequena e densa, porém tentar fatorar matrizes enormes pode ser problemático do ponto de vista numérico. O problema se agrava quando a matriz A é esparsa porque o processo de solução iria requerer a construção de uma matriz densa a partir dos elementos não nulos e operar em um dos cada n^2 elementos, inclusive os elementos nulos. Isso levaria a uma grande carga de memória e um enorme tempo de processamento.

Para obter soluções com menor gasto de recursos, é necessário utilizar métodos que levem em conta o fato da matriz ser esparsa. O número de operações de ponto flutuante deve ser dependente do número de elementos não nulos e não do tamanho da matriz.

Além disso, o esquema de armazenamento otimizado da matriz é crucial na eficiência do método numérico considerado, tanto em economia de memória quanto em operações de ponto flutuante, que impactam diretamente no tempo de processamento.

O advento das CPUs multicore e das GPUs manycores abre caminho para o desenvolvimento de softwares de alto desempenho e sistemas paralelos. O desafio agora é desenvolver sistemas computacionais que consigam utilizar de forma eficiente o número de cores disponíveis no sistema e que o desempenho destes acompanhe de forma transparente o aumento do número de núcleos disponíveis.

O objetivo deste trabalho é a criação de um sistema, provedor de primitivas de computação paralela, para a construção do Método do Gradiente Conjugado [7] para a resolução de sistemas de equações lineares esparsos de grande porte, aproveitando o poder computacional de múltiplas GPUs. A implementação do sistema desenvolvido neste trabalho utiliza a tecnologia OpenCL [8].

1.2 Trabalhos relacionados

Resoluções de sistemas de equações lineares já foram implementadas em várias plataformas multicore. Williams et al. apresentam em [13] uma panorâmica sobre o desempenho da operação de SpMV (multiplicação de matriz esparsa por vetor) em algumas arquiteturas de CPUs. Uma comparação similar foi feita por Wiggers et al. em [14], aonde o método do gradiente conjugado foi implementado em CPUs e GPUs. Implementações do gradiente conjugado e outros solvers foram apresentadas por Bolz et al. em [15], seus métodos dependem do pipeline gráfico programável. As matrizes esparsas foram armazenadas no formato compressed sparse row (CSR) juntamente com

um vetor adicional que guarda os elementos da diagonal principal.

Buatois et al. apresentam em [16] um mapeamento em GPU do método do gradiente conjugado em CUDA usando o formato block Compressed Sparse Row (BCSR) para armazenar a matriz. No entanto, como o método não foi otimizado, i.e. não houve preocupação com a organização da memória, o máximo desempenho do hardware não foi alcançado. Bell e Garland [17] propuseram vários métodos para realizar a operação de SpMV, os quais levaram em conta como a matriz estava estruturada. Eles implementaram rotinas eficientes para várias representações de matrizes, como o formato diagonal (DIA), row-packed (ELLPACK/ITPACK), coordinate list (COO), CSR, packet (PKT) e uma estrutura híbrida. Essa última se mostrou adequada para matrizes não estruturadas alcançando um melhor desempenho. Essa abordagem guarda uma parte da matriz usando ELLPACK e outra parte usando COO. É sabido que o ELLPACK se torna ineficiente se o número de elementos por linha varia muito [17,18,19]. Embora uma lista extensa de resultados ter sido apresentada, uma análise completa dos métodos não foi oferecida. No entanto, eles sugerem que uma melhor utilização de grupos de threads pode potencialmente melhorar o desempenho, mas isso foi deixado em aberto. Essa sugestão foi seguida por Monakov e Avetisyan [18]. Porém sua abordagem ocupava muita memória.

Cevahir et al. [20] propuseram uma melhora no formato Jagged Diagonals (JDS), o qual reordena a matriz de acordo com o número de elementos diferentes de zero em cada linha, e armazena o resultado de forma similar ao método CSR. Cevahir e al. apresentam em [21] uma implementação paralela do gradiente conjugado em um cluster GPU. A forma de armazenamento foi uma mistura do Jagged melhorado [20] com outros formatos do [17]. Para um dado problema, a melhor abordagem era primeiro realizar um benchmarking do desempenho de cada formato individualmente e só então selecionar o que era executado mais rápido. Isso representa claramente uma desvantagem, já que é preciso converter de modo off-line a matriz de entrava em vários formatos para então realizar os testes.

Por fim existe o ViennaCL [22], um framework que incorporou métodos de resoluções de sistemas de equações lineares em sua última versão (lançada em agosto/2012), utiliza OpenCL mas não oferece suporte ao formato Matrix Market e nem a multi-GPU.

1.3 Estrutura e organização da dissertação

O restante da dissertação está organizado da seguinte forma: O Capítulo 2 descreve a teoria completa do método do gradiente conjugado. O Capítulo 3 apresenta o conceito de GPUGPU (*General purpose computing on graphics processing units*) e também sua relação com o objetivo da dissertação. O Capítulo 4 apresenta a metodologia adotada bem como os detalhes da implementação do método do gradiente conjugado em OpenCL e seus resultados. O Capítulo 5 discute a relação entre trabalhos feitos por terceiros com a presente dissertação . Por fim as conclusões são descritas no Capítulo 6.

2
O método do gradiente conjugado

Nesse capítulo apresentamos o método do gradiente conjugado. Começamos com uma visão geral de métodos numéricos, então apresentamos o método da máxima descida que serve de base para o entendimento do método do gradiente conjugado. Utilizamos como referência para esse capítulo o artigo de Shewchuck [7].

2.1
Métodos numéricos para solução de sistemas de equações lineares

Métodos numéricos para solução de sistemas de equações lineares são divididos principalmente em dois grupos:

- Métodos Diretos: são aqueles que, exceto por erros de arredondamento, fornecem a solução exata de um sistema de equações lineares, caso ela exista, por meio de um número finito de operações aritméticas. São métodos bastante utilizados na resolução de sistemas de equações densas de pequeno a médio porte. Entende-se por sistema denso aquele no qual a matriz dos coeficientes tem um número pequeno de elementos nulos. Métodos diretos são geralmente evitados em problemas práticos que exigem a resolução de sistemas de equações lineares de grande porte porque podem apresentar problemas de desempenho e eficiência.

- Métodos Iterativos: esses métodos se caracterizam pela aplicação de um procedimento de forma repetida, ou seja, por repetir um determinado cálculo várias vezes, obtendo a cada repetição, ou iteração, um resultado mais preciso que aquele obtido na iteração anterior. Uma importante subclasse desses métodos é a dos métodos iterativos estacionários de grau um, nos quais o resultado obtido em cada iteração é uma função, somente, do resultado da iteração anterior.

Em sistemas de grande porte os erros de arredondamento de um método direto podem tornar a solução sem significado, enquanto que nos métodos iterativos os erros de arredondamento não se acumulam. Os métodos iterativos utilizam menos memória do computador, se tornando portanto vantajosos

quando a matriz dos coeficientes é uma matriz esparsa, e além disso, possuem a vantagem de se auto corrigir se um erro é cometido. Ainda podem ser usados para reduzir os erros de arredondamento na solução obtida por métodos exatos, e também, sob certas condições, serem aplicados para resolver um conjunto de equações não lineares.

2.2 O método da máxima descida (steepest descent)

Apesar de normalmente não ser usado na solução de sistemas de equações lineares, o método da máxima descida é apresentado aqui para facilitar a compreensão do método do gradiente conjugado. Primeiramente, no entanto, se faz necessária a definição de matrizes positivas definidas, para as quais estes métodos são aplicados.

2.2.1 Matrizes positivas-definidas

Em $Ax = b$, é possível pensar na matriz A como uma transformação que converte o vetor x no vetor b. Matrizes são frequentemente classificadas na forma que elas transformam os vetores, e se os dois vetores possuem exatamente o mesmo comprimento e direção, A é chamada de matriz identidade. Agora suponha que A transforma x de forma que b aponte para a direção oposta, nesse caso o produto interno $x \cdot b$ será negativo. Da mesma forma, se o produto interno for zero, então A transformou x em um vetor resultante que aponta em uma direção ortogonal. Frequentemente, as rotinas de álgebra linear requerem matrizes que nunca mudem a direção de um vetor ou produza um vetor que aponte para uma direção ortogonal. Isto é, $\bar{x}.(Ax)$ deve ser positivo para todos os x. Se uma matriz atende a esse requisito, é chamada de matriz positiva definida. Esta propriedade será de extrema importância na discussão a seguir.

2.2.2 Teoria do método da máxima descida

O objetivo do método da máxima descida é encontrar o vetor x da equação $Ax = b$. Escolhendo por exemplo x_0, como uma primeira tentativa de aproximação do vetor x, podemos validá-la calculando a subtração do resultado de $A.x_0$ do resultado de $A.x$. Se a diferença for maior que a tolerância, será preciso fazer mais tentativas. Porém não é razoável que essas tentativas sejam feitas aleatoriamente, sendo preciso um método que garanta que a próxima tentativa, x_1, será mais próxima a x do que x_0 foi. Mas como? Responder essa

questão requer uma visita ao cálculo numérico. Primeiramente deve-se obter uma função f cuja derivada em cada ponto z seja igual a $Az - b$:

$$f(z) = \frac{1}{2} z.(Az) - z.b + c$$

$$f'(z) = Az - b$$

Aqui, $b = Ax$ e c é uma constante arbitrária. Imagine que z é igual a dois vetores $x + y$. Assumindo que a matriz A é simétrica pode-se substituir $x + y$ por z e chegar no seguinte resultado:

$$f(z) = f(x) + \frac{1}{2} y.Ay$$

Se A é positiva-definida, o segundo termo é sempre maior que zero para todo y. Dessa forma $f(z)$ tem seu valor mínimo quando $z = x$. A figura 2.1 mostra $f(x)$ e $f(x_0)$, onde x_0 é a primeira tentativa de x.

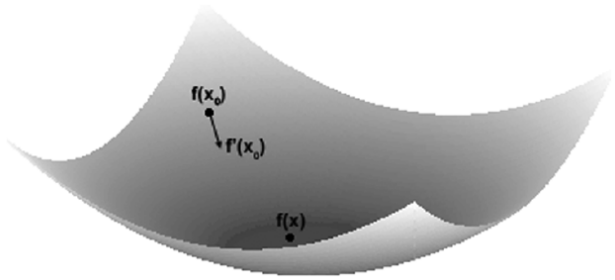

Figura 2.1: Superfície de $f(x)$

A derivada de $f(z)$ é igual a $Az - b$, então $f'(x_0)$ é igual a $Ax_0 - b$. A derivada $f'(x_0)$ identifica a direção de maior elevação da função em $z = x_0$, assim a direção da maior decaimento em $z = x_0$ é dada por $-f'(x_0)$, ou $b - Ax_0$. Essa direção é chamada de r_0, ou residual. Com o residual, é possível fazer a próxima tentativa da seguinte maneira:

$$x_1 = x_0 + \alpha r_0$$

O vetor residual r_0 diz a direção que deverá ser tomada de x_0 para x_1. O escalar é um parâmetro que varia sobre a reta de x_0 para x_1. Como o objetivo é o decaimento, é preciso escolher x_1 tal que $f(x_0 + \alpha r_0)$ seja menor que $f(z)$ em todos os outros pontos da linha. Esse valor mínimo pode ser obtido igualando $f'(z)$ a zero. Isso é dado na equação abaixo:

$$\frac{d(f(x_0 + \alpha r_0))}{d\alpha} = f'(x_0 + \alpha_0 r_0).r_0 = 0$$

Essa equação usa α_0 para denotar a distância de x_0 para x_1. Ela também mostra que, para $f(x_1)$ ser mínimo, $f'(x_1)$ deve ser ortogonal a r_1. Isto é mostrado na figura 2.2.

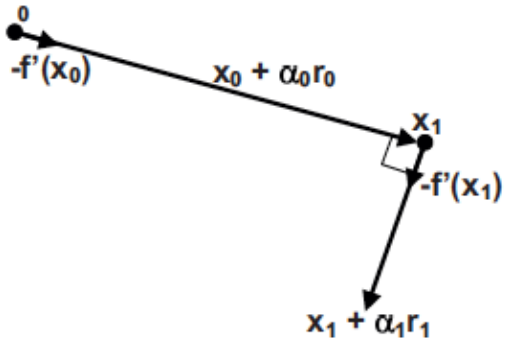

Figura 2.2: Usando o vetor residual para achar os próximos passos

Sabendo que r_1 é igual a $b - Ax_1$ e que é ortogonal a r_0 é possível determinar α_0 da seguinte maneira:

$$r_1.r_0 = 0$$

$$(b - Ax_1).r_0 = 0$$

$$(b - A(x_0 + \alpha_0 r_0)).r_0 = 0$$

$$(b - Ax_0 - \alpha_0 Ar_0).r_0 = 0$$

$$\alpha_0 Ar_0.r_0 - (b - Ax_0).r_0$$

$$\alpha_0 = \frac{r_0.r_0}{Ar_0.r_0}$$

Com essa fórmula para α_0, é possível dar o próximo passo, x_1. Esse processo pode ser repetido para os passos seguintes (x_2, x_3, x_4, etc). Para testar o erro de cada passo basta comparar o tamanho do vetor residual correspondente, $|r_i|$, com a tolerância. Não e preciso computar $r_i = b - Ax_i$ em cada passo. Ao invés disso pode-se basear cada r_i no r_{i-1} anteriormente calculado, como mostrado a seguir:

$$r_i = b - Ax_i$$

$$r_i = b - A(x_{i-1} + \alpha_{i-1} r_{i-1}) = b - Ax_{i-1} + \alpha_{i-1} Ar_{i-1}$$

$$r_i = r_{i-1} + \alpha_{i-1} Ar_{i-1}$$

A vantagem de computar r_i dessa maneira é que o produto matriz-vetor Ar_{i-1} já foi computado no processo de achar α_{i-1}. Portanto, é preciso apenas

calcular o produto matriz-vetor uma vez por iteração. Se $|r_i|$ ficar abaixo da tolerância, a resposta foi encontrada: x_i. No caso de sistemas lineares representados por matrizes esparsas simétricas, existem métodos melhores que o da máxima descida. A próxima seção discute o método do gradiente conjugado, o qual provê uma convergência muito mais rápida.

2.3 O método do gradiente conjugado

Assim como o método da máxima descida, o método do gradiente conjugado realiza uma série de tentativas para aproximar x em $Ax = b$. Essa seção discute os dois conceitos necessários ao método do gradiente conjugado — ortogonalização e conjugação — e em seguida apresenta a descrição do método.

2.3.1 Ortogonalização e o método de Gram-Schmidt

O processo de ortogonalização é iniciado com um grupo de vetores. É feita então a comparação deles em pares, e alterações são feitas até que todos sejam completamente diferentes uns dos outros. Dois vetores são completamente diferentes, ou ortogonais, se o produto interno entre eles é igual a zero.

O método de Gram-Schmidt ortogonaliza um conjunto de vetores usando projeções de vetores. Em uma revisão rápida, a projeção de um vetor b em um vetor a é a componente de b que aponta na direção de a. Essa projeção, denotada por $proj_a b$, é definida da seguinte forma:

$$proj_a b = \frac{a.b}{|a|^2} a$$

Como $proj_a b$ tem a mesma direção de a, $b - proj_a b$ deve ser ortogonal a a. Portanto, é possivel ortogonalizar dois vetores a e b computando a projeção de b em a e subtraindo a projeção de b.

Esse processo pode ser expandido para três vetores. Se um terceiro vetor, c, está incluido no espaço, os três vetores ortogonais (v_1, v_2, v_3) correspondentes a (a, b, c) podem ser calculados da seguinte forma:

$$v_1 = a_1$$

$$v_2 = b - proj_{v1} b$$

$$v_3 = c - proj_{v1} c - proj_{v2} c$$

Note que esse processo projeta c nos vetores ortogonais v_1 e v_2 e não nos vetores de entrada a e b.

Se existem n vetores de entrada, $a_1 ... a_n$ o processo de Gram-Schmidt computa o n-ésimo vetor ortogonal na forma:

$$v_n = a_n - \sum_{i=1}^{n-1} proj_{v_i} a_n$$

$$v_n = a_n - \sum_{i=1}^{n-1} \frac{v_i . a_n}{|v_i|^2} v_1 = a_n - \sum_{i=1}^{n-1} \frac{v_i . a_n}{v_1 . v_1} v_i$$

O processo de Gram-Schmidt pode produzir vetores ortogonais para qualquer número de vetores de entrada não ortogonais com a condição de todos os vetores serem *linearmente independentes*. Isto é, nenhum vetor pode ser expressado como a soma ponderada dos outros vetores. Se qualquer vetor é *linearmente dependente* aos outros, um dos vetores ortogonais será igual a zero.

2.3.2 Conjugação

Dois vetores, p e q, são *conjugados* com respeito a matriz A se $p \cdot Aq$ é igual a zero. Isto é, dois vetores são conjugados com respeito a matriz se o primeiro vetor é ortogonal ao produto da matriz e do segundo vetor. Se A é simétrica e positiva-definida, então $p \cdot Aq = q \cdot Ap = 0$.

É possível gerar um conjunto de vetores conjugados a outro usando um processo similar ao método de Gram-Schmidt. Isto é mostrado na equação seguinte:

$$v_n = a_n - \sum_{i=1}^{n-1} \frac{v_i . Aa_n}{v_i . Av_i} v_i$$

Essa relação entre os vetores é importante na solução de sistemas de matrizes esparsas utilizando o método do gradiente conjugado.

2.3.3 Teoria do método do gradiente conjugado

O método do gradiente conjugado tem muito em comum com o método da máxima descida. Em ambos os casos o objetivo é fazer tentativas, x_i, que levam de x_0 para x. Ambos os métodos utilizam o vetor residual, r_i, para julgar quão longe a tentativa corrente está da resposta correta.

A primeira diferença entre os métodos é que, enquanto o método da máxima descida usa r_i como sendo a direção de x_i para x_{i+1}, o método do gradiente conjugado computa um novo vetor, p_i. A direção inicial, p_0, é inicializada como sendo r_0, porém cada direção subsequente será *conjugada* à direção predecessora. Na forma de equações tem-se:

$$p_0 = r_0$$

$$p_i = r_i - \sum_{j=0}^{i-1} \frac{p_j.Ar_j}{p_j.Ap_j} p_j$$

Tendo escolhido uma direção, as direções subsequentes serão geradas de maneira similar à usada no método da máxima descida:

$$x_{i+1} = x_i + \alpha_i p_i$$

$$\alpha_i = \frac{r_i.r_i}{p_i.Ap_i}$$

O algoritmo como um todo requer os oito passos a seguir:

1 Fazer o palpite inicial, x_0.

2 Computar o primeiro vetor residual e a direção inicializando r_0 e p_0 iguais a b.

3 Computar a distância de x_i para x_{i+1}, $\alpha_i = \frac{r_i \cdot r_i}{p_i \cdot Ap_i}$

4 Determinar a próxima tentativa, $x_{i+1} = x_i + \alpha_i p_i$.

5 Computar o próximo vetor residual, $r_{i+1} = r_i - \alpha_i Ap_i$.

6 Computar a próxima direção, $p_{i+1} = r_{i+1} + (\frac{r_{i+1} \cdot r_{i+1}}{r_i \cdot r_i}) p_i$.

7 Achar $|r_{i+1}|$. Se este for menor que a tolerância, por ex. (0.01), o objetivo foi alcançado.

8 Repetir os passos 3 até 7.

3
GPGPU

Esse capítulo começa introduzindo o conceito de programação genérica em GPU (GPGPU). Em seguida explicamos os aspectos técnicos da arquitetura OpenCL para que seja possível entender o processo de implementação do método do gradiente conjugado em GPU apresentado no capítulo 4.

3.1
Introdução

Movida pela demanda de mercado por aplicações em tempo real, como gráficos 3D de alta definição, a GPU evoluiu para um processador manycore altamente paralelo com um gigantesco poder de computação e uma grande largura de barramento de memória.

A razão por trás da grande diferença de poder de computação entre a CPU e a GPU é que a GPU é especializada para tarefas de computação intensiva e portanto foi projetada de forma que mais transistores são dedicados para processamento ao invés de utilizá-los para cache e/ou controle de fluxo. Veja a 3.1 para uma representação gráfica.

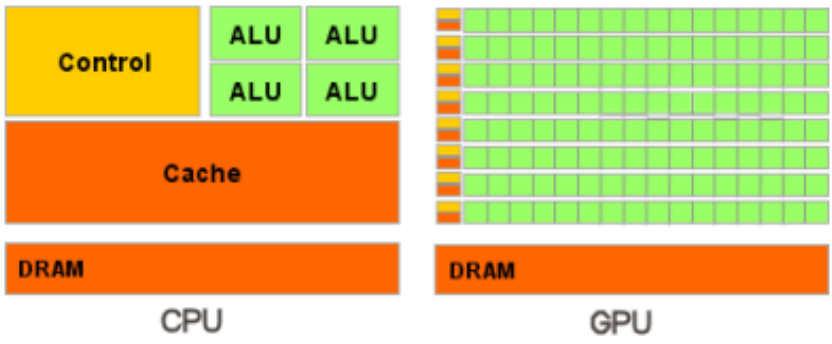

Figura 3.1: GPU versus CPU arquitetura [23]

Mais especificamente, a GPU foi especialmente projetada para resolver problemas que podem ser expressos em termos de computações com diversos dados em paralelo sendo executadas por grupos de threads em *lock-step*. Todas as threads que pertencem ao mesmo grupo (chamado de *wavefront* em

OpenCL) executam a mesma instrução mas em diferentes porções de dados (modelo SIMD - *single instruction multiple data* - ou ainda SIMT - *single instruction multiple thread*). Por isso existe pouca necessidade de uma unidade de controle de fluxo sofisticada e a latência de acesso a memória pode ser ocultada com cálculos (sobreposição da comunicação com a computação) ao invés de caches gigantes.

Devido ao custo relativamente baixo de placas de vídeo, essas estão presentes na maioria dos computadores da atual geração, dando a possibilidade de áreas diversas, como processamento de imagem e vídeo, simulação física, computação financeira e até biologia computacional, terem seus algoritmos acelerados pelo processamento de dados em paralelo na GPU. Estas GPUs são formadas por unidades de processamento chamadas multiprocessadores, com registradores e memória compartilhada próprios, capazes de executar milhares de threads. Isso deu início a uma nova abordagem, a *GPU Computing* (também chamada *GPGPU - General purpose computing on graphics processing units*), conceito que visa explorar as vantagens das placas gráficas modernas em aplicações altamente paralelizáveis que exigem intenso fluxo de cálculos. Com seu crescente potencial no futuro dos sistemas computacionais, várias aplicações têm sido identificadas e mapeadas com sucesso para execução em GPUs [9].

Embora disponível desde 2002, foi somente em 2007, com o lançamento oficial da plataforma CUDA, que a GPU Computing ganhou notoriedade e passou a ser largamente utilizada na computação de alto desempenho.

3.2 OpenCL

A plataforma CUDA oferece aos programadores uma interface para resolver problemas arbitrários onde a manipulação dos dados pode ser paralelizada. No entanto, CUDA está restrita a placas NVIDIA, e também é preciso usar um compilador especial chamado NVCC, o que pode dificultar a integração entre CUDA e projetos pré-existentes.

O Khronos Group (que promove a especificação do OpenGL) apresentou uma especificação para o OpenCL (Open Computing Language) 1.0 [8] que foi lançado então no final de 2008. O OpenCL fornece um padrão multi-plataforma para a criação de programas paralelos em ambientes heterogêneos (CPUs, GPUs e outros tipos de processadores). A presença do padrão OpenCL torna possível realizar cálculos genéricos na GPU de forma independente da plataforma e do fabricante.

OpenCL oferece oportunidades interessantes para desenvolvedores de

software que desejam acelerar o desempenho de suas aplicações. No entanto, é importante estar ciente que a placa de vídeo não é a resposta para tudo. As placas de vídeo ajudam a resolver problemas que envolvem grande quantidade de computação e preferencialmente não dependam de precisão dupla. Além disso, executar um problema na GPU envolve uma sobrecarga própria, e somente é vantajoso para classes específicas de aplicações.

A figura 3.2 ilustra as relações entre o Host (CPU) e Devices (dispositivos que executam o código OpenCL C, como CPUs, GPUs e aceleradores). O host executa código escrito em C#, C++, Visual Basic, Java, C. Envia informações, comandos de execução e lê os dados dos dispositivos. Os dispositivos executam código OpenCL. Existe um compilador OpenCL específico para a CPU, para a GPU e para as placas aceleradoras. Através da API de OpenCL podemos identificar os dispositivos, compilar os programas e enviar e receber informações.

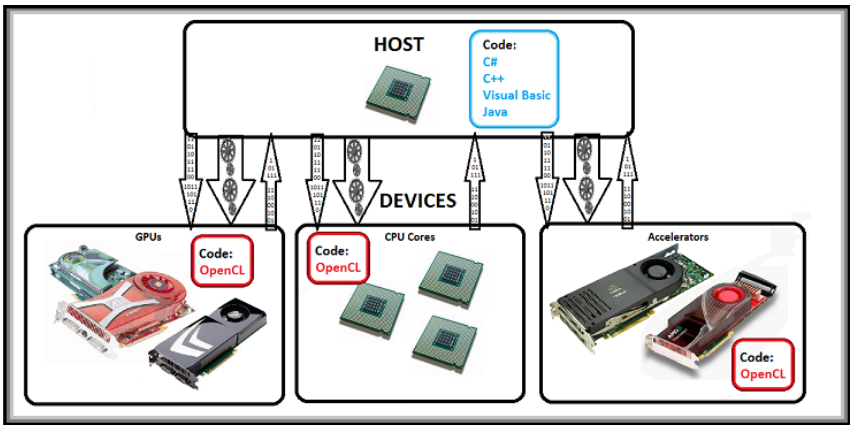

Figura 3.2: Esquema Host/Device OpenCL [24]

3.2.1 Aspectos técnicos de OpenCL

Uma das primeiras características de OpenCL é a possibilidade de executar aplicações usando milhares e milhares de threads, chamadas *itens de trabalho*. Para fazer uso desse poder de processamento é preciso entender como os itens de trabalho acessam a memória e como a sincronização pode ser utilizada para coordenar essa operação. Para atingir esse entendimento é importante discutir como configurar dois parâmetros: o tamanho global e tamanho local, em aplicações de grande escala. Como será demonstrado adiante, encontrar os valores certos para esses parâmetros é de extrema importância no desempenho de processamento do kernel.

O limite superior do número de itens de trabalho que é possível gerar é o máximo valor de `size_t` (veja `SIZE_MAX` em stdint.h [30]). O número total

de itens de trabalho criados é chamado *tamanho global*. A plataforma OpenCL tem as seguintes características:

- Os itens de trabalho são divididos em *grupos de trabalho*. Cada grupo de trabalho tem sua própria identificação numérica chamada *identificador de grupo*.
- Todo item de trabalho possui dois identificadores. Seu identificador de grupo o identifica dentro de todos os outros itens de trabalho gerados que executarão o kernel. Seu identificador local o identifica apenas dentro de seu próprio grupo de trabalho.
- Cada grupo de trabalho possui seu próprio bloco de memória chamado memória local. Para muitos dispositivos, o tempo de acesso de um item de trabalho a essa porção de memória é muito mais rápido do que o tempo de acesso à memória global.
- Para conservar a largura de banda, as operações que acessam a memória global são combinadas em uma operação para o grupo de trabalho inteiro.
- Itens de trabalho em um mesmo grupo de trabalho podem ser sincronizados com chamadas da função *barrier*. OpenCL não provê nenhuma função para sincronizar itens de trabalho em diferentes grupos de trabalho.

Como o acesso a memória global consome muito tempo, muitos kernels a acessam apenas duas vezes: a primeira para ler os dados de entrada (ocasionalmente copiando parte para a memória local) e outra para escrever os resultados da memória local para a memória global. Nessa abordagem, todo o processamento é feito nos registradores de cada um dos itens de trabalho (chamado de memória privada) e/ou na memória local.

Para utilizar a memória local da melhor forma, é importante ter o máximo possível de itens de trabalho em um grupo de trabalho. O número de itens de trabalho em um grupo de trabalho é chamado de *tamanho local*. OpenCL provê um método direto para achar o tamanho local máximo para o par kernel/dispositivo. Na subseção 4.3.1 será explicado como esse método funciona.

Na terminologia de OpenCL, tarefas são chamadas de *kernels*. Um kernel é uma função especialmente codificada que é executada por um ou mais dispositivos OpenCL. Kernels são enviados a seu respectivo dispositivo ou dispositivos pelas aplicações host. Uma aplicação host é uma aplicação convencional C/C++ sendo executada na CPU, que recebe o nome de host. Uma aplicação OpenCL é estruturada sob a forma de uma aplicação

host contendo chamadas que disparam a execução de kernels em diferentes dispositivos. Aplicações host gerenciam seus dispositivos conectados usando um tipo abstrato chamado de *contexto*.

Para executar um kernel, o host seleciona a função kernel desejada depois associa os argumentos necessários. Este pacote (função mais argumentos) é enviado para uma estrutura chamada *fila de comando*. A fila de comando é o mecanismo através do qual o host envia o trabalho ao dispositivo.

3.2.2 Modelo de memória

O modelo de memória do OpenCL define cinco regiões de memória:

- Memória host - Visível apenas para o host. Ela serve como uma memória de transação entre o host e os dispositivos.
- Memória global - Permite o acesso de leitura/escrita a todos os itens de trabalho em todos os grupos de trabalho. As leituras e escritas na memória global podem utilizar a "cache"dependendo das capacidades do dispositivo.
- Memória constante - Região de memória pertencente a memória global que permanece constante durante a execução de um kernel. Os itens de trabalho tem apenas permissão de leitura a essa região.
- Memória local - Região de memória local ao grupo de trabalho. Pode ser usada para alocar variáveis compartilhadas por todos os itens de trabalho de um determinado grupo de trabalho.
- Memória privada - Região de memória privada ao item de trabalho. Variáveis definidas na memória privada de um item de trabalho não são visíveis a outros itens de trabalho.

As regiões de memória e suas respectivas interações são exemplificadas na figura 3.3. Os itens de trabalho são executados nos elementos processadores (PEs) e cada elemento processador tem sua memória privada associada. Um grupo de trabalho é executado em uma unidade de computação (*compute unit*) e cada unidade de computação tem sua memória local associada que é compartilhada por todos os itens de trabalho do grupo que está sendo executado.

O host tem acesso apenas a memória global do dispositivo OpenCL, todas as outras são transparentes para o host. A cópia de dados da memória host para a memória global ocorre de maneira explícita. O host enfilera comandos

para transferir dados entre as regiões (host/global) tanto para escrita como para leitura.

Quando falamos em execução concorrente, o modelo de memória precisa definir cuidadosamente como os objetos de memória interagem com o kernel e o host. Esse é o problema da consistência de memória. OpenCL não estipula o modelo de consistência de memória no host. Dessa forma fica a cargo do programador tomar o cuidado necessário. A seguir, vamos avaliar como isso pode ser feito nas regiões de memória do dispositivo OpenCL.

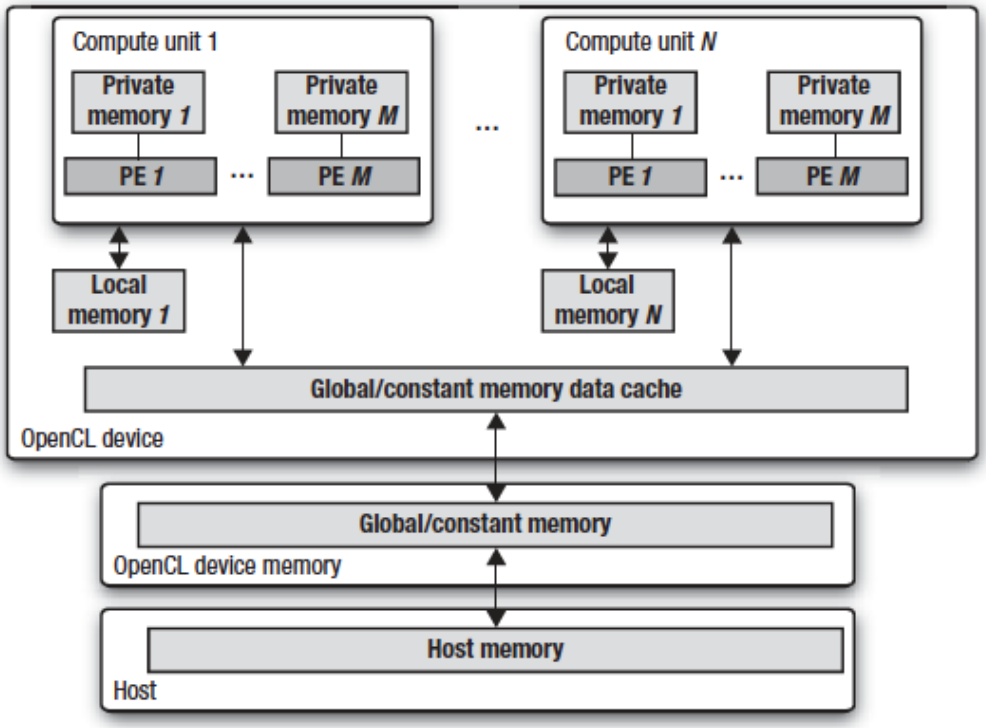

Figura 3.3: O modelo de memória em OpenCL e como as diferentes regiões de memória interagem entre si [27]

A memória privada é visível apenas a um item de trabalho. Ela segue o modelo de memória padrão load/store semelhante à adotada na programação sequencial. Já na memória local, os valores vistos pelos itens de trabalho de um determinado grupo de trabalho devem ter sua consistência garantida através de pontos de sincronização. Por exemplo, uma barreira requer que todas as operações de escrita tenham sido concluídas por todos os itens de trabalho no grupo de trabalho antes de prosseguir pela barreira. Como a memória local é compartilhada apenas dentro de um grupo de trabalho isso é o suficiente para garantir a consistência da memória. Para a memória global, a consistência também pode ser garantida através de barreiras, porém, apenas para os itens de

trabalho dentro de um grupo de trabalho. A memória global é compartilhada entre todos os grupos de trabalho e não há nenhuma forma de sincronismo ou garantia de consistência da memória global entre os grupos de trabalho. OpenCL define um modelo de consistência relaxado [26], isto é, um valor visto na memória global por um determinado item de trabalho não tem garantia de ser o mesmo para todos os outros itens de trabalho em todas as vezes pois a coerência/consistência dos dados só é requerida em certos pontos do fluxo de execução.

3.2.3 Aplicação host

O primeiro passo na programação de qualquer aplicação OpenCL é codificar a aplicação host. Para lidar com a configuração geral da aplicação, OpenCL define seis estruturas de dados: *plataformas*, *dispositivos*, *contextos*, *programs*, *kernels* e *filas de comando*.

3.2.4 Plataformas

O desenvolvimento de qualquer sistema OpenCL requer o conhecimento do hardware disponível. Por exemplo, o computador a ser utilizado pode possuir uma ou mais placas de vídeo e cada placa pode ser de um fabricante diferente. O conjunto de dispositivos pertencentes a um mesmo fabricante recebe o nome de plataforma. Através da aplicação host é possível identificar as plataformas disponíveis bem como contabilizar o número de dispositivos OpenCL existentes. Conhecendo-se o número de dispositivos pode-se, por exemplo, distribuir as tarefas entre eles.

3.2.5 Dispositivos

Associado a cada plataforma, existe um conjunto de dispositivos que a aplicação utiliza para executar código. Dada uma plataforma, um lista de dispositivos suportados pode ser descoberta. Em tempo de execução é possível escolher um ou mais dispositivos de preferência para realizar uma determinada tarefa.

3.2.6 Contextos

Um contexto é uma abstração que possibilita associar dispositivos a objetos de memória (buffers, imagens) e a filas de comandos (que provê uma

interface entre um contexto e um determinado dispositivo). É o contexto que efetua a comunicação com, e entre, dispositivos específicos. Atualmente os dispositivos em um contexto devem ser de uma mesma plataforma, isto é, não é possível criar um contexto contendo um dispositivo AMD e outro NVIDIA, sendo preciso criar diferentes contextos para cada plataforma. Uma aplicação host pode gerenciar dispositivos usando mais de um contexto e pode criar ainda múltiplos contextos para dispositivos em uma mesma plataforma. A figura 3.4 ilustra dois contextos (um com um CPU e uma GPU e outro com apenas uma GPU) criados dentro de uma das plataformas disponíveis.

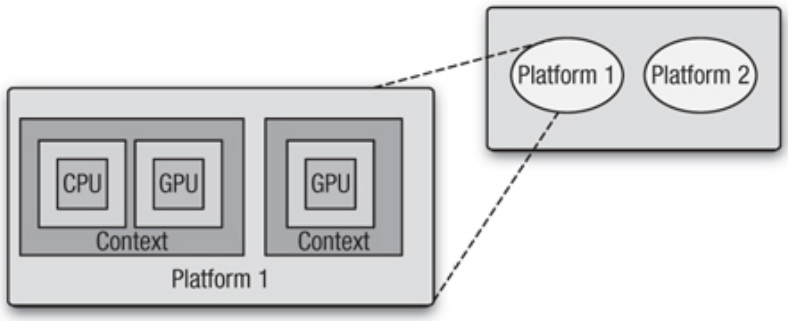

Figura 3.4: Contextos em OpenCL [27]

3.2.7 Programas e kernels

As aplicações OpenCL expressam as funções que serão executadas em paralelo nos dispositivos na forma de kernels. Kernels são escritos na linguagem OpenCL C e sua assinatura deve ser precedida da diretiva `__kernel`. Para passar argumentos para uma função kernel, a aplicação deve criar um objeto kernel. Objetos kernels podem ser operados usando funções da API que permitem a passagem de argumentos ou até mesmo a extração de informações do kernel. Os objetos kernels são criados a partir de objetos programas. Os objetos programas contêm a coleção de funções kernels que são definidas no código fonte do programa. O objetivo principal do objeto programa é facilitar a compilação dos kernels nos dispositivos que os executarão, já que existe essa ligação explícita kernel-dispositivo na criação da aplicação. Uma analogia para entender a diferença entre objetos kernels e objetos programas é que o objeto programa funciona como uma biblioteca dinâmica que guarda uma coleção de funções kernel.

3.2.8 Fila de comandos

A interação entre o host e os dispositivos OpenCL ocorre através dos comandos postados pelo host nas filas de comandos. A fila de comandos é criada pelo host e é associada a um único dispositivo após o contexto ser definido. Cada dispositivo precisa ter obrigatoriamente pelo menos uma fila de comandos, podendo ter mais. O OpenCL suporta três tipos de comandos:

- Comandos de execução do kernel - executam o kernel no dispositivo.
- Comandos de memória - transferem dados entre o host e os objetos de memória, movem dados entre objetos de memória, mapeiam objetos de memória a partir do espaço de endereço do host.
- Comandos de sincronização - restringem a ordem de execução dos comandos.

  Quando múltiplos kernels são submetidos a uma fila, eles podem precisar interagir. Por exemplo, um conjunto de kernels pode gerar objetos de memórias que outro conjunto de kernels precisam manipular. Nesse caso, comandos de sincronização podem ser usados para forçar que o primeiro conjunto de kernels termine a sua execução antes do próximo conjunto começar a executar. Para dar suporte a esse mecanismo de sincronização, os comandos submetidos a uma fila de comandos geram objetos evento. Uma fila de comandos pode esperar até uma certa condição em um objeto evento existir.

3.3 Desempenho de CUDA vs OpenCL

OpenCL é uma linguagem portável para programação de alto desempenho multiplataforma. Diferentemente do kernel CUDA, o kernel OpenCL pode ser compilado em tempo de execução. Esse estilo de compilação *just-in-time* permite que o compilador gere código mais otimizado para o dispositivo. CUDA, por sua vez, é desenvolvido pela mesma companhia que desenvolve o hardware no qual CUDA é executado, logo é esperado que sua execução seja mais rápida nesses dispositivos. Considerando esses fatores, é interessante comparar o desempenho de OpenCL em relação a CUDA em aplicações do mundo real.

Nos testes disponíveis em [31], CUDA obteve um melhor desempenho na transferência de dados de/para a GPU. Os kernels de CUDA também foram executados mais rápidos que os de OpenCL. Os kernels testados foram: FFT - Transformada rápida de Fourier, DGEMM - subrotina de multiplicação

de matriz por vetor presente na biblioteca BLAS (*Basic Linear Algebra Subroutine*) [32], MD - simulação dinâmica de moléculas, S3D - solver direto para a equação de Navier-Stokes. CUDA parece ser a melhor escolha para aplicações aonde atingir o maior desempenho possível seja importante desde que se tenha a disponibilidade de um dispositivo compatível.

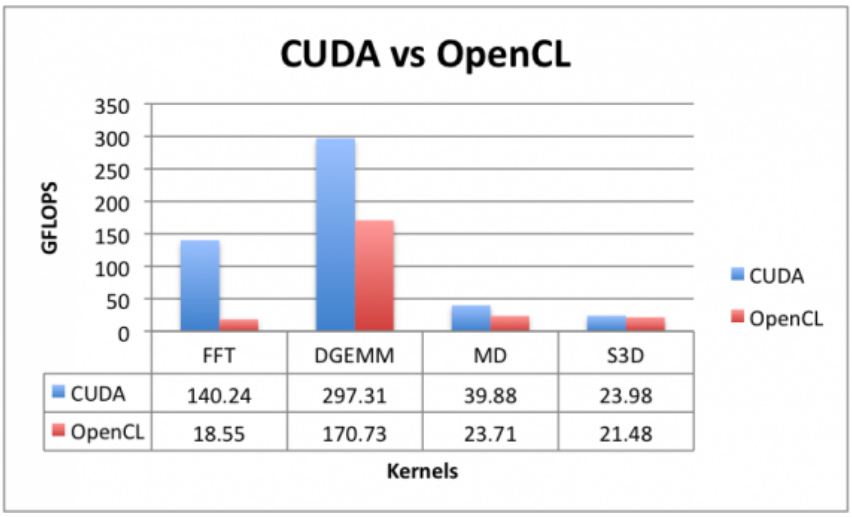

Figura 3.5: CUDA vs OpenCL [31]

4
O método do gradiente conjugado em OpenCL

O sistema implementado permite resolver sistemas de equações lineares esparsos de grande porte através da exploração do potencial computacional de múltiplas GPUs, utilizando a tecnologia OpenCL. Para efetuar a resolução, o método do gradiente conjugado é subdividido em kernels. Esse capítulo detalha todo o processo de implementação e consequentes automatizações implementadas para atingir um bom desempenho do método independente do conjunto de placas de vídeo que estiver sendo utilizado.

O método do gradiente conjugado, apresentado no capítulo 2, é uma abordagem iterativa amplamente utilizada para resolver sistemas lineares do tipo $Ax = b$. Para uma matriz M x M simétrica positiva definida, o método converge em até M iterações. Os oito passos do método são apresentados abaixo:

1 Fazer o palpite inicial, x_0.

2 Computar o primeiro vetor residual e a direção de busca inicializando r_0 e p_0 iguais a b.

3 Computar a distância de x_i para x_{i+1}, $\alpha_i = \frac{r_i \cdot r_i}{p_i \cdot Ap_i}$

4 Determinar a próxima tentativa, $x_{i+1} = x_i + \alpha_i p_i$.

5 Computar o próximo vetor residual, $r_{i+1} = r_i - \alpha_i Ap_i$.

6 Computar a próxima direção de busca, $p_{i+1} = r_{i+1} + (\frac{r_{i+1} \cdot r_{i+1}}{r_i \cdot r_i})p_i$.

7 Achar $|r_{i+1}|$. Se este for menor que a tolerância, por ex. (0.01), o objetivo foi alcançado.

8 Repetir os passos 3 até 7.

4.1 Armazenamento de matrizes esparsas no formato Matrix Market

Matrizes esparsas grandes não devem ser armazenadas em arrays bi-dimensionais. Isto é devido ao fato delas poderem conter centenas de zeros para cada elemento não nulo. Logo é melhor armazenar apenas os elementos não nulos e a localização destes na matriz. Neste trabalho o método de armazenamento empregado foi o formato definido pelo repositório Matrix Market [10]. Esse repositório oferece um conjunto de matrizes de domínio público usados para modelar alguns tipos de aplicações. A maioria das matrizes esparsas oriundas de problemas de física ou engenharia são são da ordem de centenas e milhares de linhas e colunas. Todas as matrizes que foram usadas para validar a presente implementação são parte do repositório Matrix Market patrocinado pelo NIST (*National Institute for Standards and Technology*).

4.1.1 Acesso aos dados no arquivo Matrix Market

Os arquivos matrix market possuem extensão .mtx. A informação contida nesse arquivo é simples de ser acessada e consiste em 3 partes:

- Banner - identifica o formato da matriz e a natureza da matriz (real ou complexa, simétrica ou geral).
- Informação de tamanho - identifica as dimensões da matriz e o número de elementos não nulos.
- Dados - a linha, coluna e o valor de cada elemento não nulo da matriz. O formato linha-coluna-valor é chamado formato coordenado.

Por exemplo, as primeiras 5 linhas do arquivo *bcsstk05.mtx* são:

```
%%MatrixMarket matrix coordinate real symmetric
153 153 1288
1 1  3.1431392791300e+05
4 1 -8.6857870528200e+04
5 1  5.6340240342600e+04
```

O banner mostra que a matriz é real, simétrica e que seus valores são armazenados no formato coordenado. A próxima linha mostra que as dimensões da matriz são 153x153 e que o arquivo provê 1288 elementos não nulos. O primeiro elemento não nulo está localizado na linha 1, coluna 1 e o segundo está localizado na linha 4, coluna 1. Quando a matriz é simétrica, o arquivo matrix market contém apenas seus elementos não nulos da diagonal e acima

dela. Esses arquivos podem ser lidos utilizando funções regulares de C/C++, porém o NIST provê suas próprias rotinas de I/O no arquivo *mmio.c* [11]. Esse código é de domínio público.

4.2 Divisão do método em kernels ("building blocks")

O método do gradiente conjugado foi subdividido em kernels. Cada função kernel poderia ser implementada no código do programa principal, porém para melhorar a modularidade, manutenção e entendimento do código optou-se por programar cada kernel em um arquivo separado. A seguir, mostramos a implementação de cada um dos cinco kernels implementados, que juntos, compõem o método.

4.2.1 Produto interno

Considere dois vetores: $A = (a_1, a_2, ..., a_n)$ e $B = (b_1, b_2, ..., b_n)$. O produto interno entre esses dois vetores é $A \cdot B = a_1 b_1 + a_2 b_2 + ... + a_n b_n$.

O somatório $\sum_{k=1}^{n} a_k b_k$ é um caso particular de redução, que consiste em transformar um vetor de dados em apenas um elemento, agregando os dados com determinada operação. No caso do produto interno, a operação de agregação é a soma.

O algoritmo de redução desenvolvido neste trabalho foi baseado no artigo da AMD disponível em [12] e se chama redução em multi estágios. A Figura 4.1 ilustra como funciona o kernel do produto interno usando a redução em multi estágios.

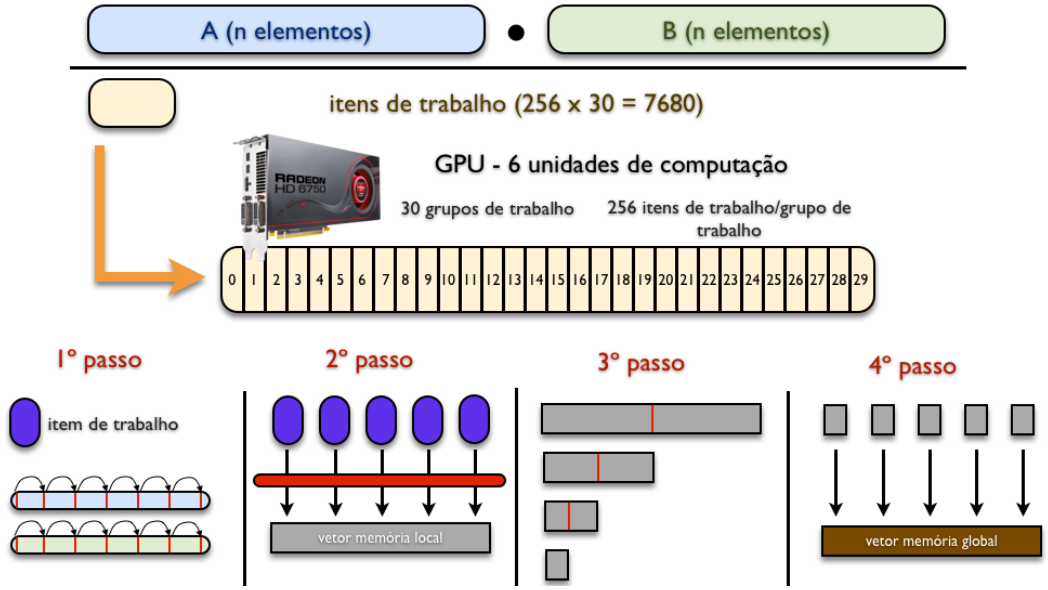

Figura 4.1: Kernel do produto interno utilizando redução em multi estágios

Considere, por exemplo, que a placa de vídeo usada tenha 6 unidades de computação. Essa informação é especialmente importante na escolha inicial do número de grupos de trabalho. No exemplo 7680 itens de trabalho foram criados constituindo 30 grupos de trabalho de 256 itens de trabalho cada.

O passo a passo implementado consiste em:

. 1º passo: cada item de trabalho acessa a memória global onde estão armazenados os vetores A e B e executa:

$$sum = A[global\_id] * B[global\_id]$$

$$sum \stackrel{+}{=} \sum_{k=global\_id}^{(n-7680)+global\_id} A[k + 7680] * B[k + 7680]$$

armazenando o resultado da variável sum em sua memória privada.

Nesse passo os itens de trabalho estão acessando a memória global, i.e. as posições dos vetores A e B em ordem, como exemplificado na figura 4.2.

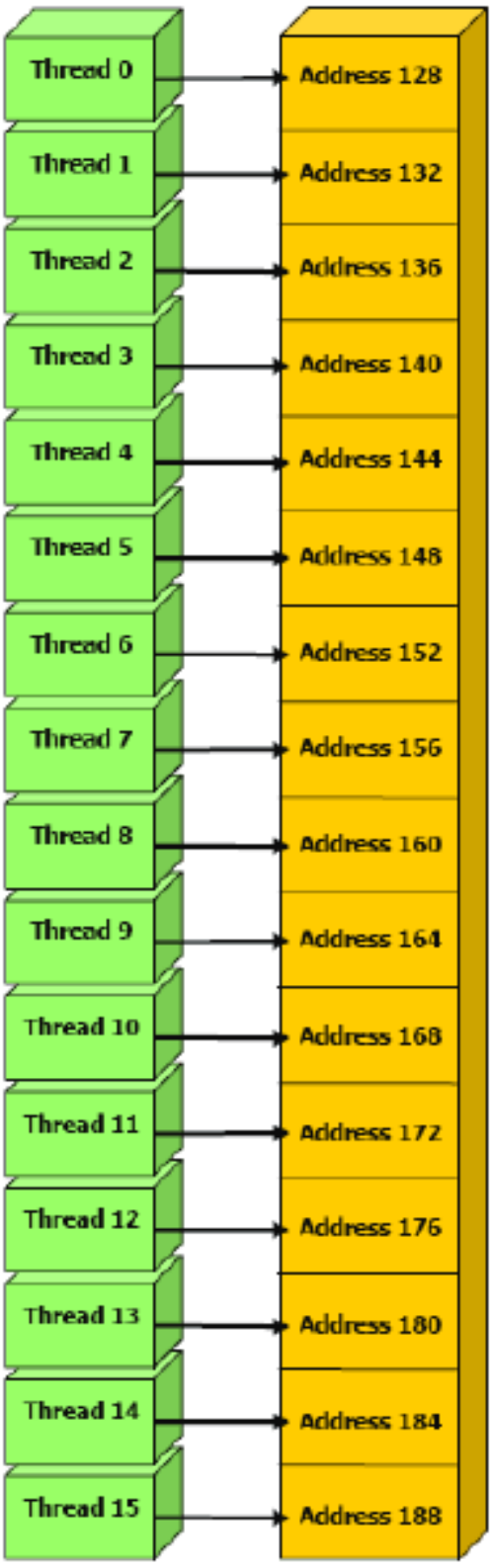

Figura 4.2: Acesso coalescido a memória global [28]

A largura de banda da memória global é mais eficientemente usada quando acessos simultâneos à memória por itens de trabalho de um mesmo grupo de trabalho (durante a execução de uma instrução de leitura ou escrita) podem ser

coalescidos em uma única transação de memória. Os itens de trabalho devem acessar palavras na sequência: o item de trabalho k deve acessar a palavra k.

- 2º passo: cada item de trabalho copia a soma de sua memória privada para o vetor residente na memória local do dispositivo. Observe que cada grupo de trabalho terá seu próprio vetor exclusivo residente na memória local.
- 3º passo: a cada iteração, cada item de trabalho da primeira metade de cada grupo de trabalho realiza a adição de seu valor com outro da segunda metade. Isso é exemplificado na figura 4.3. A cada nova iteração o vetor é reduzido pela metade e por fim cada grupo de trabalho terá apenas um elemento resultante.

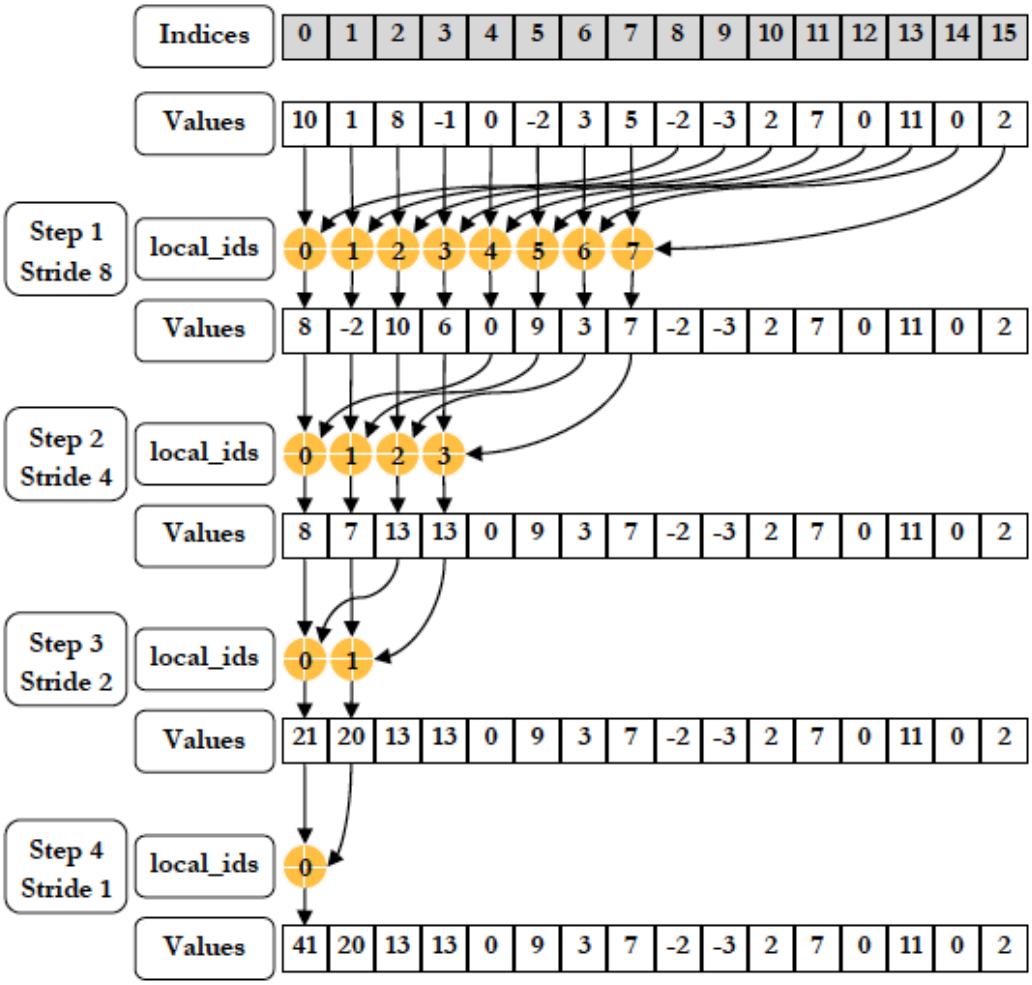

Figura 4.3: Redução em paralelo na memória local do dispositivo [29]

- 4º passo: ao final do 3º passo serão obtido 30 elementos (um para cada grupo de trabalho). O primeiro item de trabalho de cada grupo de trabalho copia seu valor calculado para a memória global do dispositivo onde pode ser transferido para a CPU que efetuará a soma dos 30 elementos.

4.2.2 Atualizações de variáveis

. Próxima tentativa

Cada item de trabalho acessa a memória global onde estão armazenados os vetores x e p e executa:

$$for(k = global\_id;\ k < n;\ k\ +=\ 7680)\ x[k]\ +=\ \alpha * p[k]$$

. Próximo residual

Cada item de trabalho acessa a memória global onde estão armazenados os vetores r e A_times_p e executa:

$$for(k = global\_id;\ k < n;\ k\ +=\ 7680)\ r[k]\ -=\ \alpha * A\_times\_p[k]$$

. Próxima direção

Cada item de trabalho acessa a memória global onde estão armazenados os vetores p e r e executa:

$$for(k = global\_id;\ k < n;\ k\ +=\ 7680)\ p[k]\ =\ r[k] + \beta * p[k]$$

Para um detalhamento completo da implementação consultar a figura 4.4.

4.2.3 Multiplicação de matriz por vetor

Cada item de trabalho acessa a memória global onde estão armazenados os vetores A_times_p, $cols$ e a matriz A e executa:

$$for(k = global\_id;\ k < n;\ k\ +=\ 7680)\ A\_times\_p[k]\ +=\ A[k] * p[cols[k]]$$

Para um detalhamento completo da implementação consultar a figura 4.5.

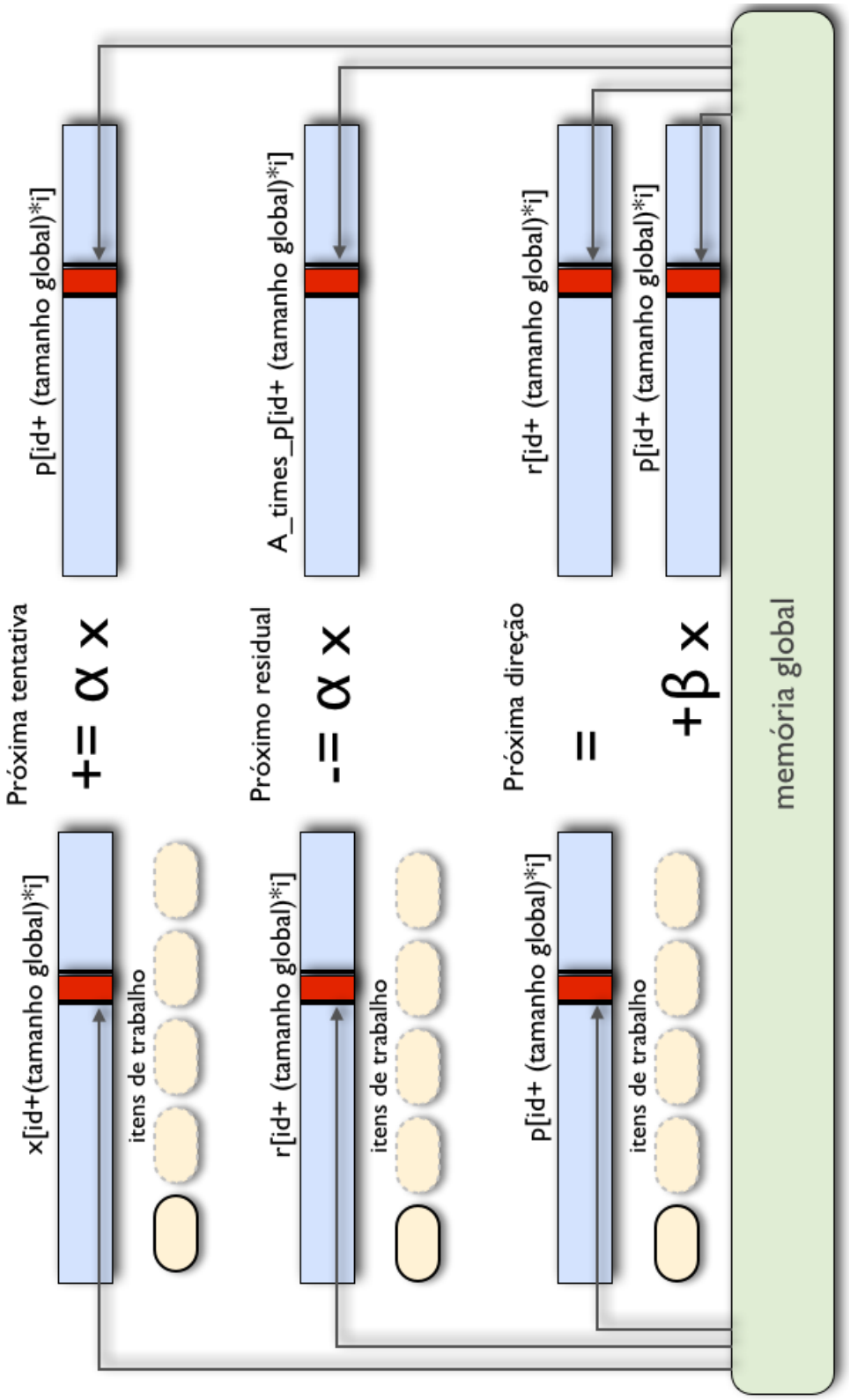

Figura 4.4: Esquema de como foram implementados os kernels responsáveis pela atualização de variáveis.

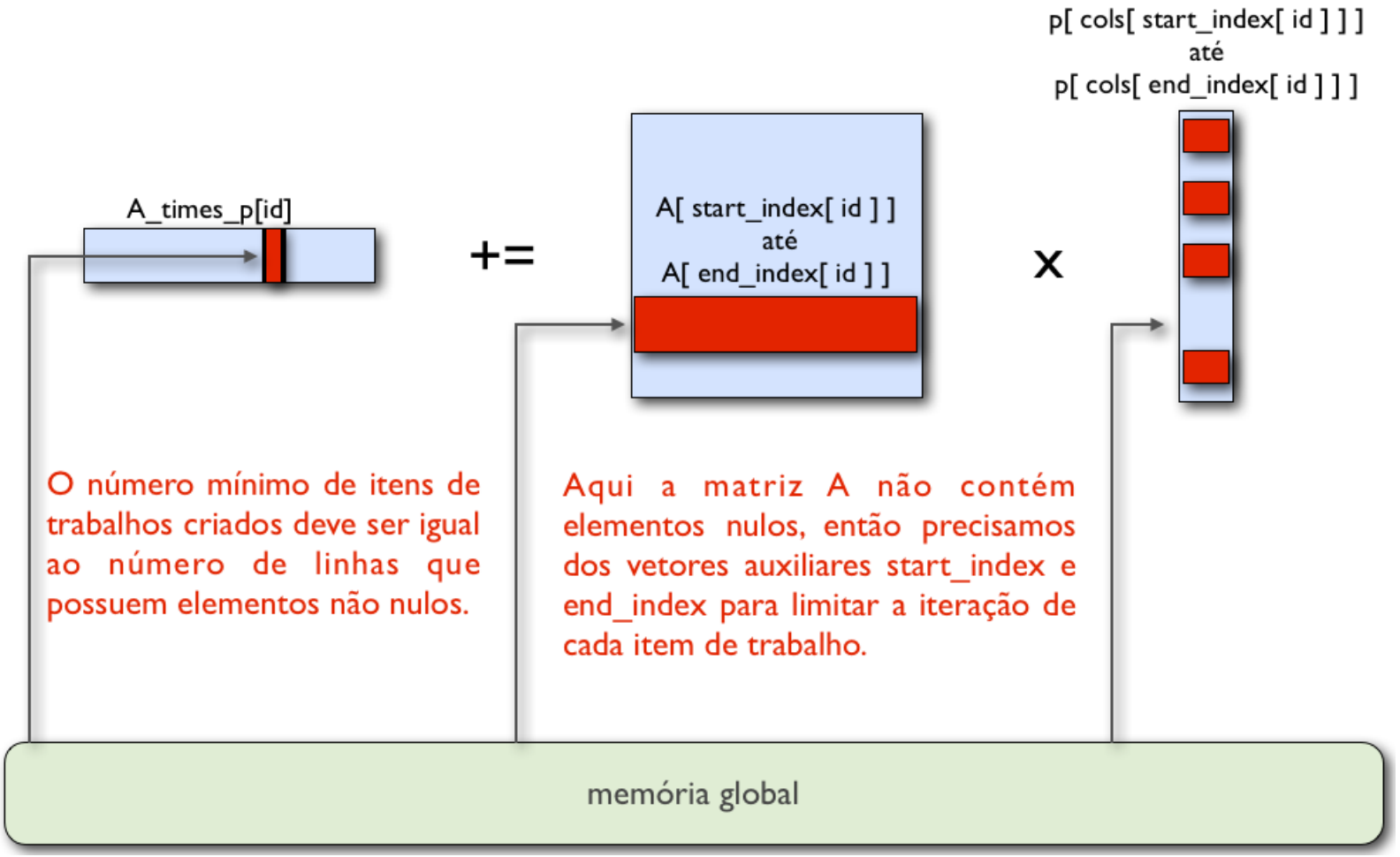

Figura 4.5: Esquema de como foram implementados o kernel de multiplicação matriz por vetor.

4.3 Configuração

4.3.1 Obtenção do tamanho máximo de um grupo de trabalho

Não importa quantos itens de trabalho são gerados para um kernel, o número máximo de itens de trabalho em um grupo de trabalho será o mesmo. Se o tamanho global exceder o número máximo de itens de trabalho em um grupo de trabalho, OpenCL irá criar grupos de trabalho adicionais. Uma aplicação pode gerar um número praticamente ilimitado de grupos de trabalho, porém o número de grupos de trabalho que serão executados em paralelo é limitado pelo número de unidades de computação presentes no dispositivo.

O tamanho máximo de um grupo de trabalho depende de dois fatores: os recursos fornecidos pelo dispositivo e o recursos requeridos pelo kernel. Normalmente, os recursos inerentes à criação de um item de trabalho são a memória local e a memória privada disponível. Quanto mais memória um kernel requer, menos itens de trabalho serão disponibilizados para executá-lo. Os itens de trabalho podem acessar a memória global/constante ao invés da memória local/privada, porém a largura de banda cai consideravelmente.

Determinar a utilização de recursos de um kernel pode ser tedioso e passível de erros. Em OpenCL isso pode ser realizado através da função `clGetKernelWorkGroupInfo`. Ao chamar essa função passando como um de seus parâmetros `CL_KERNEL_WORK_GROUP_SIZE`, o tamanho máximo do grupo de trabalho para um determinado par kernel/dispositivo é retornado. A tabela 4.1 mostra o valor retornado nos dispositivos utilizados. Para garantir a máxima utilização do dispositivo na execução de cada grupo de trabalho deverá ser adotado o número correspondente na tabela.

Tabela 4.1: Tamanho local

Dispositivo	Tamanho local (itens de trabalho)
ATI Radeon HD 6750M	256
NVIDIA Tesla C1060	512

4.3.2 Escolha do número de grupos de trabalho

O próximo passo importante é saber quantas unidades de computação cada dispositivo tem. Essa informação é crucial na escolha do número de grupos de trabalho que será criado. Isso garante uma utilização mais eficiente do dispositivo.

Cada grupo de trabalho é executado em uma unidade de computação e cada unidade de computação executa apenas um grupo de trabalho por vez. Esse relacionamento é mostrado graficamente na figura 4.6.

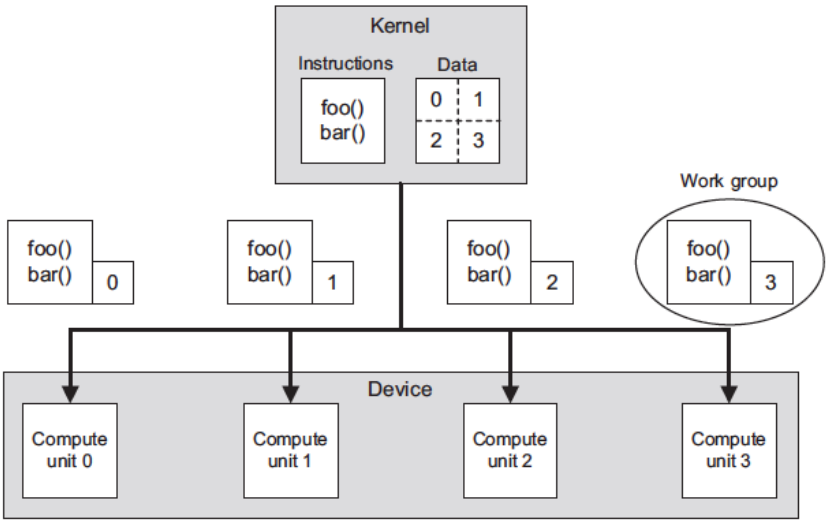

Figura 4.6: Enviando grupos de trabalho para as unidades de computação [27]

OpenCL oferece uma função, *clGetDeviceInfo*, que fornece diversas informações sobre o dispositivo dependendo da flag que é passada como parâmetro. Para que o retorno da função seja o número de unidades de computação disponíveis no dispositivo requisitado, o valor do parâmetro a ser passado é `CL_DEVICE_MAX_COMPUTE_UNITS`. A tabela 4.2 mostra o valor retornado nos dispositivos utilizados neste trabalho.

Tabela 4.2: Número de unidades de computação em cada dispositivo

Dispositivo	Unidades de computação
ATI Radeon HD 6750M	6
NVIDIA Tesla C1060	30

O controle do número de grupos de trabalho criado é feito através da divisão do tamanho global pelo tamanho local. Esses valores são passados para a função *clEnqueueNDRangeKernel* quando o kernel é enfileirado para execução. O tamanho local é definido como mostrado na seção 4.3.1. Para determinar o valor ótimo do tamanho global é preciso experimentar diferentes valores e ver qual produz um menor tempo de execução. Se o tamanho local for `k` e o número de unidades de computação for `n`, este número de grupos de trabalho parece uma boa tentativa inicial, i.e.: tamanho global = `k * n` itens de trabalho. O ajuste fino varia de acordo com o dispositivo e o kernel que será executado. Um número bom de grupos de trabalho para um determinado par

dispositivo/kernel não necessariamente será igual para outro. Por isso se faz necessária uma avaliação individual de execução em cada um deles. A tabela 4.3 mostra o valor inicial, tamanho global, para cada dispositivo utilizado como exemplo.

Tabela 4.3: Tamanho global inicial

Dispositivo	Tamanho global inicial (itens de trabalho)
ATI Radeon HD 6750M	1536
NVIDIA Tesla C1060	15360

4.3.3 Automatização da escolha do número de grupos de trabalho

Para estimar o valor ótimo do número de grupos de trabalho usamos o valor inicial mostrado na tabela 4.3 (um grupo de trabalho por unidade de computação). A função de configuração realiza uma série de tentativas, partindo do valor base na tabela 4.3 e incrementando um grupo de trabalho por unidade de computação a cada nova tentativa até haver uma saturação na progressão temporal. O número que obtiver o melhor tempo será o escolhido. Cada sequência completa de testes é feita para cada par dispositivo/kernel individualmente. A figura 4.7 ilustra todo o processo de testes implementado. A função, *cgKernelProfiler*, que implementamos, recebe como argumento o kernel a ser analisado e também a dimensão do dado de entrada do kernel, i.e. a dimensão da matriz, e retorna o número de itens de trabalho por grupo de trabalho que obteve o menor tempo de execução. A função *cgKernelProfiler* é totalmente independente do tipo de dispositivo e fabricante, bastando apenas que o dispositivo seja compatível com OpenCL.

Observando os resultados dos testes e em especial a tabela 4.4 percebe-se o impacto da escolha certa do número de grupos de trabalho no tempo final de execução do kernel (e consequentemente do método).

Além disso, é possível constatar que uma vez ultrapassado o número de itens de trabalho concorrentes em execução, criar mais itens de trabalho não traz nenhuma melhora no desempenho. Em muitos casos quando se tem itens de trabalho suficientes para esconder a latência (normalmente latência da memória global e/ou da ALU) não se ganha mais desempenho com mais itens de trabalho e como constatado, o desempenho ainda piora.

Essa motivação permite a projeção da automatização desses testes em não somente um dispositivo mas em um ambiente multi dispositivo. É preciso

pensar primeiramente na arquitetura do código que será implementado. Na subseção 4.4 discutiremos como essa implementação foi feita.

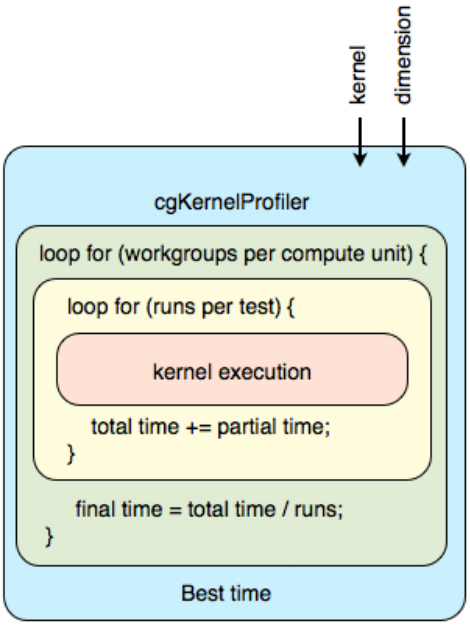

Figura 4.7: Automatização da escolha do número de grupos de trabalho

4.3.4 Resultados

As figuras 4.8, 4.9, 4.10, 4.11 e 4.12 têm por objetivo mostrar como a redução de tempo de execução varia de acordo com o aumento do número de grupos de trabalho por unidade de computação em um dispositivo. Foi considerada a dimensão 10^7 para a matriz A.

Tabela 4.4: Maior redução de tempo alcançada [ordem 10^7] — ATI Radeon HD 6750M

Kernel	Tempo 1 GT/UC	Melhor tempo	GT/UC	Redução
produto interno	0,008733	0,002177	5	75%
próxima direção	0,013816	0,005337	16	61%
próximo residual	0,013817	0,005331	16	61%
próxima tentativa	0,013791	0,005344	16	61%
matriz vetor	0,052663	0,021661	14	58%

Adicionalmente a tabela 4.4 mostra um resumo das melhores configurações de grupos de trabalho por unidade de computação que

resultaram nos melhores tempos de execução (média de 10 rodadas), para cada um dos kernels, e sua consecutiva porcentagem de redução.

Ao analisar as figuras 4.8, 4.9, 4.10, 4.11 e 4.12 podemos notar um fator comum entre todas elas: existe uma queda significativa de tempo (primeiro "mínimo local") nas primeiras variações de grupos de trabalho por unidades de computação e nas variações conseguintes há uma estabilização da variação gerando pouca ou nenhuma melhora no tempo de execução. Os kernels "produto interno", "próxima direção", "próximo residual"e "próxima tentativa"atingem o primeiro "mínimo local"com cinco grupos de trabalho por unidades de computação. Já o kernel "matriz vetor"chega mais rápido a esse primeiro mínimo, com apenas três. Dessa forma podemos parar de procurar uma melhor configuração após o primeiro mínimo local ser atingido garantindo mais rapidez na execução inicial do sistema.

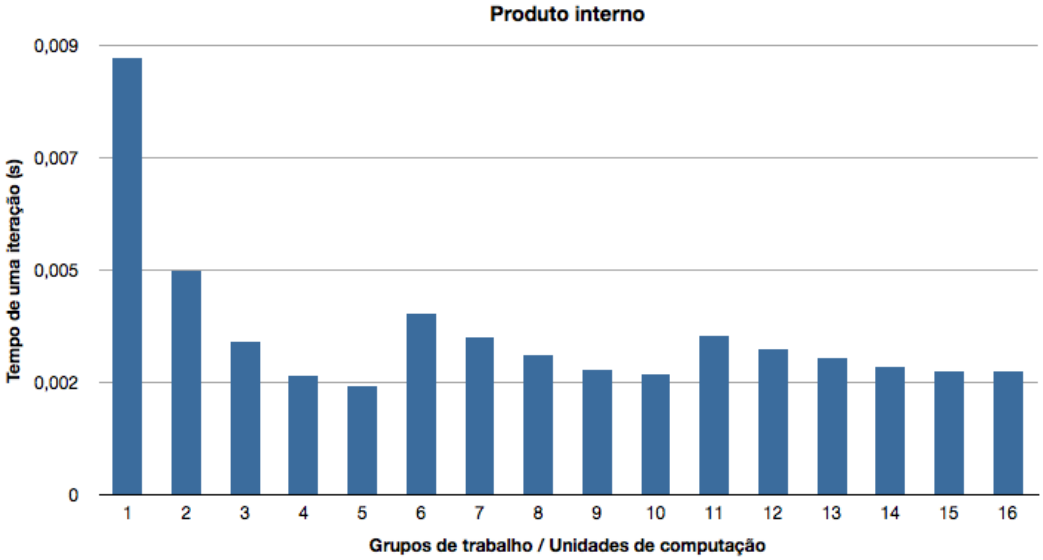

Figura 4.8: Gráfico do tempo de execução de uma iteração do kernel "produto interno", para um problema de ordem 10^7, de acordo com o número de grupos de trabalho por unidades de computação. Dispositivo: ATI Radeon HD 6750M com 6 unidades de computação.

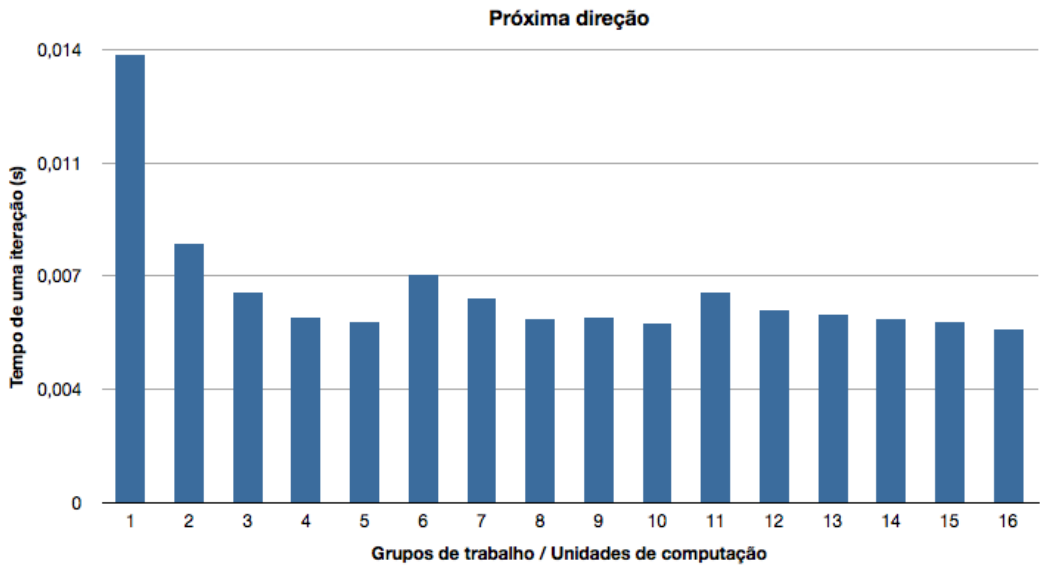

Figura 4.9: Gráfico do tempo de execução de uma iteração do kernel "próxima direção", para um problema de ordem 10^7, de acordo com o número de grupos de trabalho por unidades de computação. Dispositivo: ATI Radeon HD 6750M com 6 unidades de computação.

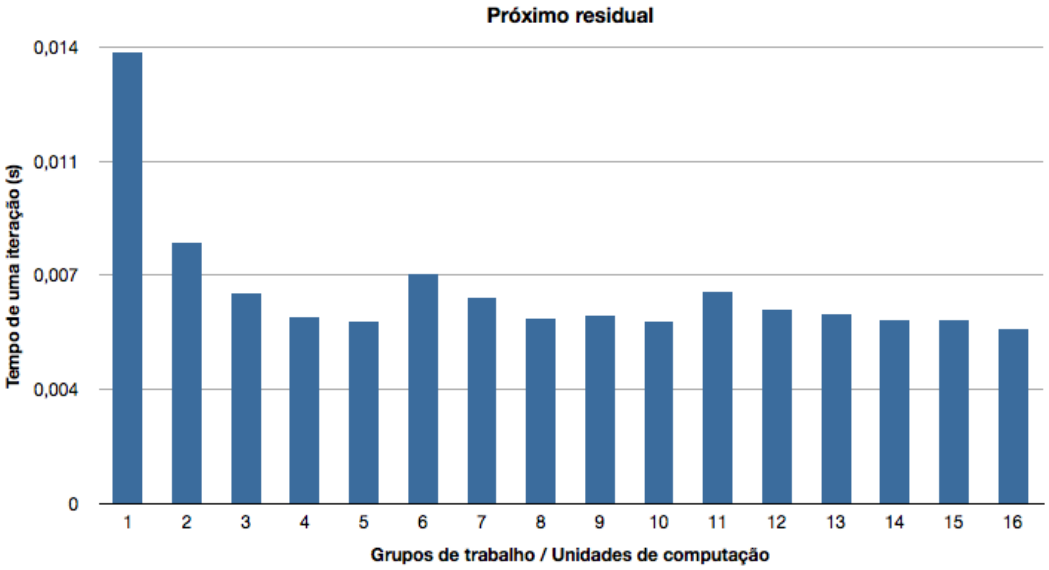

Figura 4.10: Gráfico do tempo de execução de uma iteração do kernel "próximo residual", para um problema de ordem 10^7, de acordo com o número de grupos de trabalho por unidades de computação. Dispositivo: ATI Radeon HD 6750M com 6 unidades de computação.

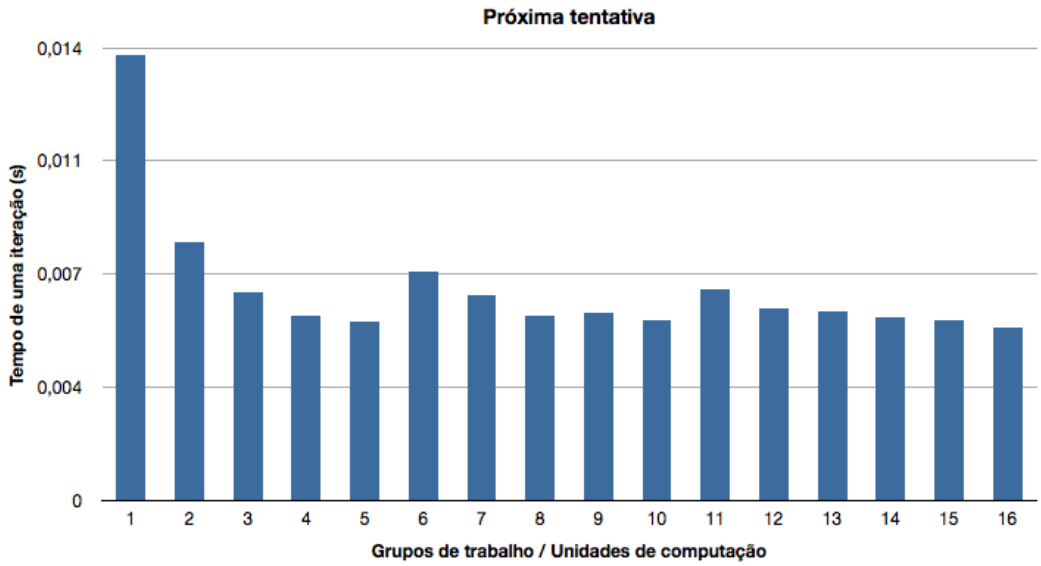

Figura 4.11: Gráfico do tempo de execução de uma iteração do kernel "próxima tentativa", para um problema de ordem 10^7, de acordo com o número de grupos de trabalho por unidades de computação. Dispositivo: ATI Radeon HD 6750M com 6 unidades de computação.

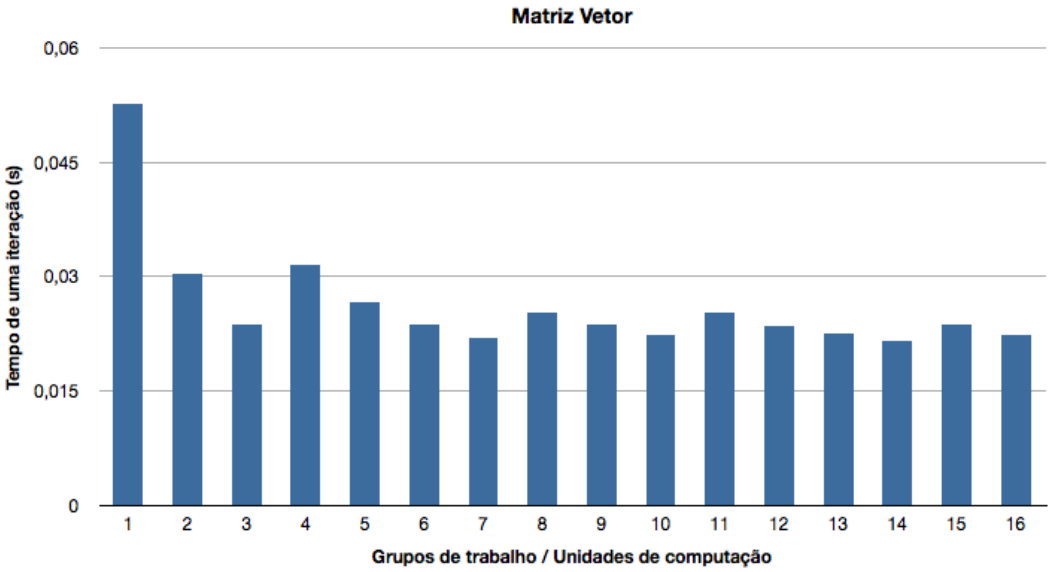

Figura 4.12: Gráfico do tempo de execução de uma iteração do kernel "matriz vetor", para um problema de ordem 10^7, de acordo com o número de grupos de trabalho por unidades de computação. Dispositivo: ATI Radeon HD 6750M com 6 unidades de computação.

4.4 Extensão do método para um ambiente multi dispositivo

A distribuição de processamento em sistemas multiprocessadores ou multicomputadores, ou seja, a divisão de trabalho entre processadores diferentes localizados no mesmo sistema ou em sistemas diferentes, deve observar os seguintes aspectos:

1 Identificação das unidades de processamento - Em sistemas multiprocessadores é necessário cuidado com a identificação das unidades de processamento de cada trabalho, ou seja, é preciso garantir que cada fragmento do trabalho seja univocamente identificado, permitindo o controle preciso de seu envio e retorno, bem como determinação da situação do processamento global a qualquer instante.

Em um ambiente multi dispositivo três cenários são possíveis: todos os dispositivos iguais, todos diferentes ou um cenário híbrido. Por esse motivo, quando se deseja que um esquema genérico seja implementado, é preciso pensar em uma estratégia que lide com qualquer ambiente.

O esquema implementado é ilustrado na figura 4.13. Dois tipos abstratos de dados foram criados:

- LocalGlobalSize: guarda os valores ótimos do tamanho local e global encontrado através da função *cgKernelProfiler*.
- AmbientInfo: guarda além do número de dispositivos um vetor do tipo LocalGlobalSize para cada dispositivo avaliado.

A função *cgMainProfiler* primeiramente analisa o ambiente em que será executada a função *cgKernelProfiler*. As informações relevantes são: o número de dispositivos diferentes que estão disponíveis e quantas unidades de computação tem em cada um deles. Se o ambiente só possuir dispositivos idênticos, apenas um será considerado para testes já que os resultados serão os mesmos. A partir daí cada dispositivo será submetido aos testes presentes na função cgKernelProfiler para cada um dos kernels. Ao final da execução da função cgMainProfiler será retornado o vetor *newAmbientInfoVector*, um vetor do tipo abstrato *AmbientInfo* e de tamanho igual ao número de kernels que forem avaliados i.e. cada posição guarda uma struct *AmbientInfo* referente ao kernel avaliado.

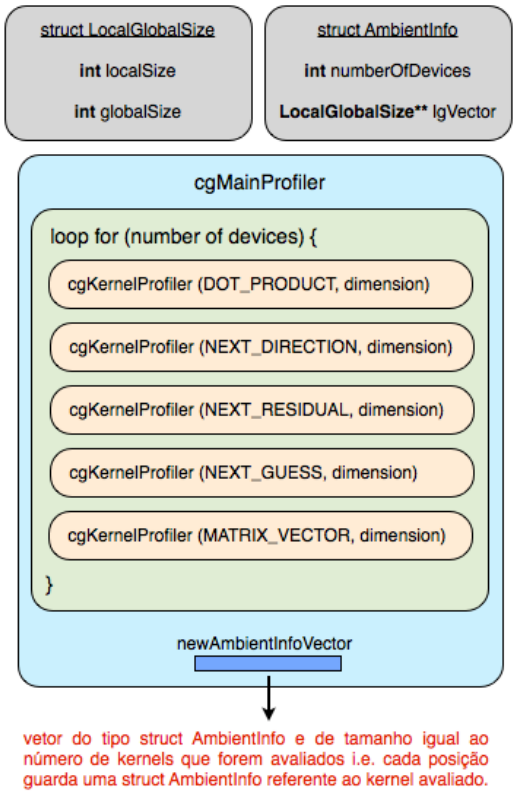

Figura 4.13: Automatização da escolha do número de grupos de trabalho em um ambiente multi dispositivo

2 Divisão do processamento - Para que o processamento possa ser distribuído, é requisito fundamental que o trabalho computacional que deve ser realizado possa ser dividido em partes cujo processamento possa se dar de maneira concorrente, ou seja, cada parte deve poder ser processada de forma autônoma, independente da ordem de submissão, processador alocado ou tempo de processamento. Desta maneira, tanto o código a ser executado (algoritmo) como os dados a serem processados, devem ser preparados para possibilitar tal divisão.

A divisão do trabalho foi feita levando em consideração o grafo de dependência gerado a partir da ordem de execução dos kernels que compõem o método do gradiente conjugado. A figura 4.14 apresenta o grafo de dependências.

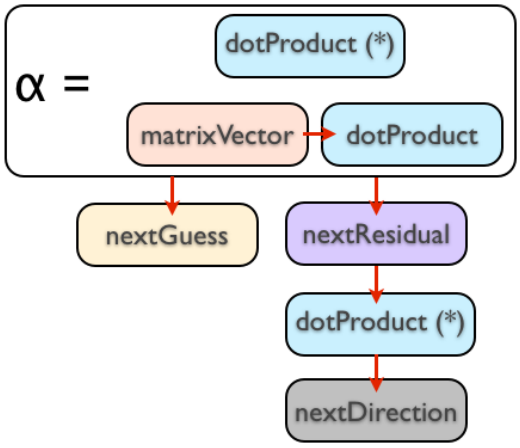

Figura 4.14: Grafo de dependência dos kernels que compõe o método do gradiente conjugado

Observe que a cada iteração é necessário calcular o valor de α que, como visto no capítulo 2, é o tamanho do próximo passo a ser dado pelo método. Só após esse valor ser calculado as próximas etapas do método podem ser executadas. O asterisco no produto interno serve para salientar que o cálculo do resultado do produto interno presente no numerador de α só é calculado na primeira iteração. A partir da segunda iteração o valor utilizado para ele será o mesmo do resultado obtido ao calcular o produto interno após o kernel do próximo residual.

As figuras 4.15, 4.16, 4.17, 4.18 e 4.19 mostram a divisão dos dados de entrada entre os dispositivos para cada um dos kernels.

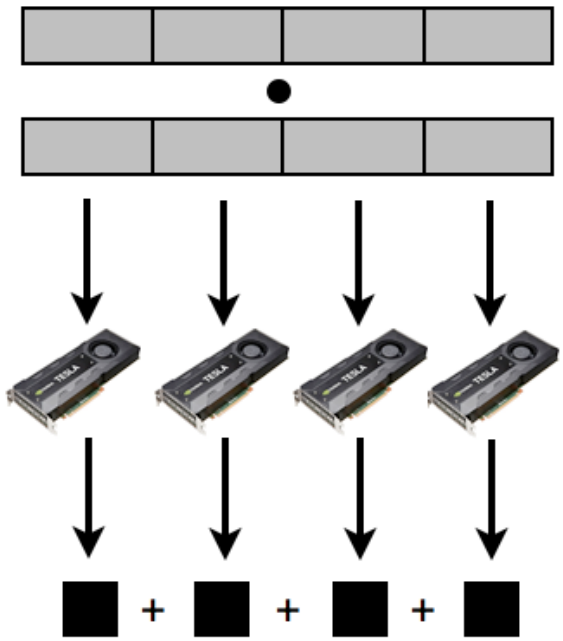

Figura 4.15: Divisão dos vetores de entrada do kernel "produto interno"entre os dispositivos disponíveis. Ao final, o host soma todas os valores resultantes gerando o resultado final

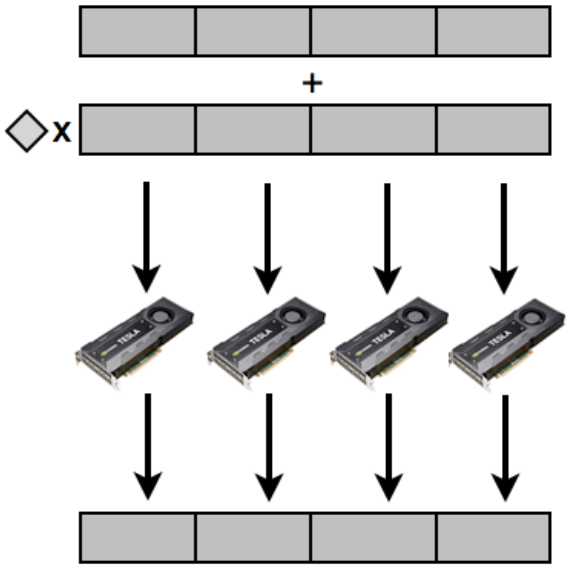

Figura 4.16: Divisão dos vetores de entrada do kernel "próxima direção"entre os dispositivos disponíveis

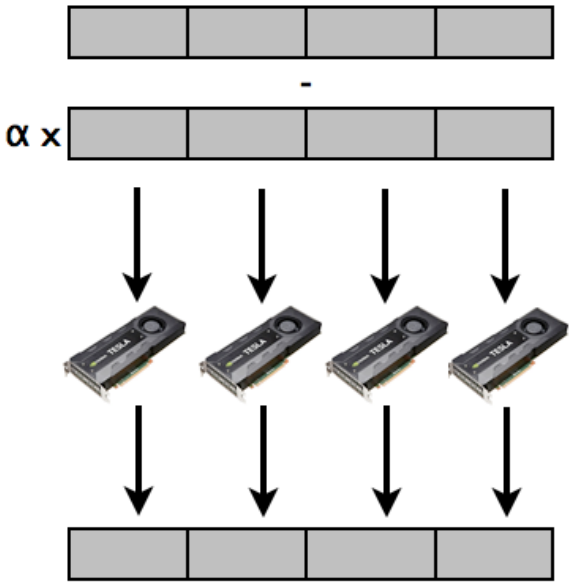

Figura 4.17: Divisão dos vetores de entrada do kernel "próximo residual"entre os dispositivos disponíveis

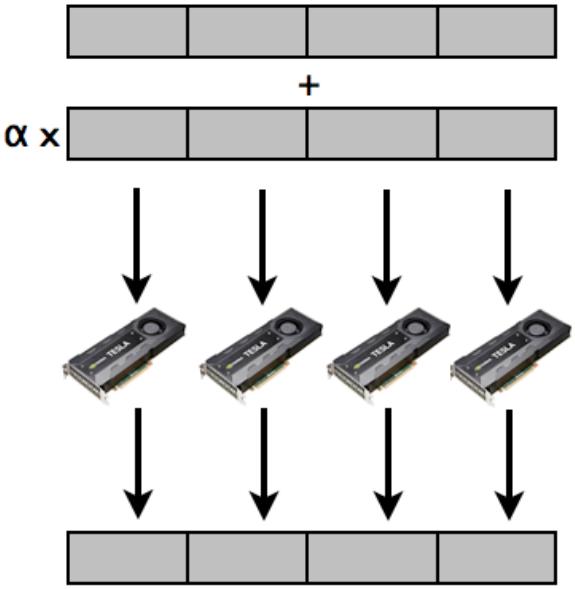

Figura 4.18: Divisão dos vetores de entrada do kernel "próxima tentativa"entre os dispositivos disponíveis

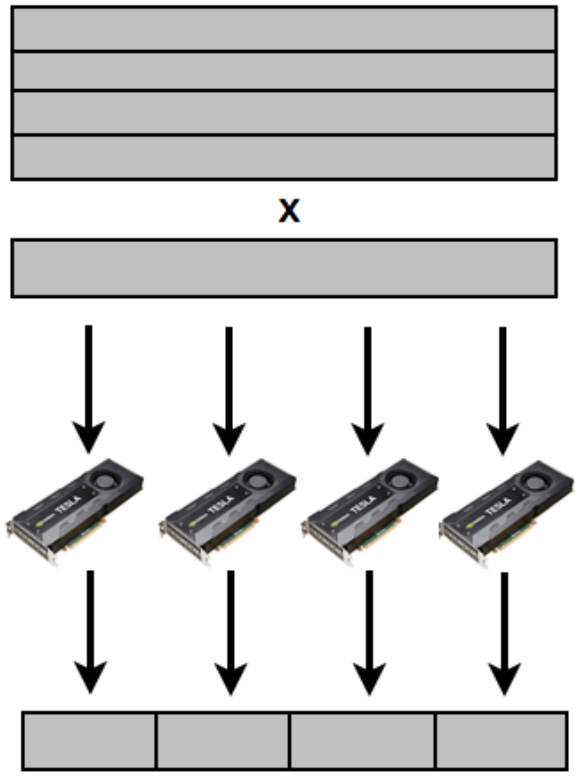

Figura 4.19: Divisão dos vetores de entrada do kernel "matriz vetor"entre os dispositivos disponíveis. Cada dispositivo receberá, além da parte correspondente da matriz, uma cópia completa do vetor multiplicador

3 Políticas de distribuição - Considerando que o desempenho de cada dispositivo pode ser substancialmente diferente, a distribuição de unidades de processamento deve, sempre que possível, distribuir o trabalho de forma diretamente proporcional à velocidade de processamento de cada dispositivo, buscando assim o melhor desempenho global.

Pensando nisso resolvemos testar uma forma de quantificar a velocidade de processamento de trabalho de cada dispositivo. Com base nas características disponíveis para consulta de cada dispositivo escolhemos multiplicar o número de unidades de computação pela frequência de clock (de cada unidade de computação). Infelizmente essa estratégia não trouxe bons resultados. Uma possível causa é encontrada no livro *OpenCL Programming Guide* no seguinte trecho:

> "OpenCL só garante que os itens de trabalho dentro de um grupo de trabalho sejam executados simultaneamente (compartilhando recursos do processador do dispositivo). No entanto, você nunca pode assumir que grupos de trabalho ou invocações de kernels são executados simultaneamente. Elas de fato, muitas vezes executam simultaneamente, mas o design do algoritmo não pode depender disso."

Isso significa que qualquer técnica de divisão implementada (partição de grupos de trabalho entre as unidades de computação de uma GPU) dependerá dessa limitação da linguagem (falta de garantia de execução concorrentes de grupos de trabalho) e isso foge do controle do programador. Dessa forma optamos por dividir o trabalho de forma igualitária entre os dispositivos. A figura 4.20 a divisão utilizada.

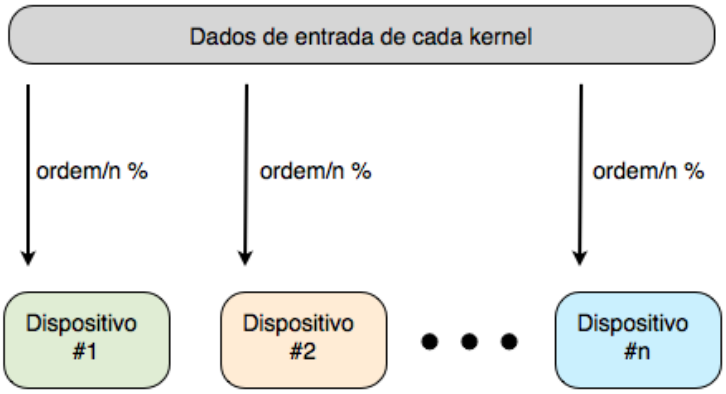

Figura 4.20: Divisão dos dados de entrada para cada dispositivo baseado no número de dispositivos.

4.4.1 Resultados

Foram utilizados dois conjuntos de placas para testar o sistema implementado.

O primeiro chamado "MultiGPU 1", disponível no laboratório Tecgraf, é composto de:

. Três placas NVIDIA Tesla C1060, cada uma com 30 unidades de computação e frequência de clock de 1296 MHz.

O segundo chamado "MultiGPU 2", disponível no laboratório Tecgraf, é composto de:

. Duas placas NVIDIA Tesla C2075, cada uma com 14 unidades de computação e frequência de clock de 1147 MHz.

As tabelas 4.5 e 4.6 mostram o tempo de execução de 100 iterações do método do gradiente conjugado (detalhando o tempo individual de cada um dos kernels) para cada uma das plataformas utilizadas. Adicionalmente o tempo de sobrecarga, i.e. tempo total menos o tempo de execução somado de todos os kernels, também é mostrado.

A figura 4.21 apresenta uma representação gráfica das tabelas 4.5 e 4.6. O tempo de sobrecarga é maior nas plataformas multi-GPU do que em apenas uma GPU, isso é devido ao acumulo de latência de comunicação de todas as placas a cada envio/recebimento de informações. Apesar disso, a latência é compensada pelo ganho de velocidade na resolução dos kernels.

Tabela 4.5: Tempo de 100 iterações do método [Ordem 10^7] — ATI Radeon HD 6750M, NVIDIA Tesla C1060, NVIDIA Tesla C2075

Kernel	Radeon HD 6750M(s)	Tesla C1060(s)	Tesla C2075(s)
Prod. interno[2x]	0,43	0,29	0,26
Próx. direção	0,53	0,23	0,20
Próx. residual	0,53	0,23	0,20
Próx. tentativa	0,53	0,23	0,20
Matriz vetor	2,16	0,69	0,57
Sobrecarga	0,26	0,13	0,12
Tempo total	4,46	1,81	1,55

Tabela 4.6: Tempo de 100 iterações do método [Ordem 10^7] — MultiGPU 1, MultiGPU 2

Kernel	MultiGPU 1(s)	MultiGPU 2(s)
Prod. interno[2x]	0,07	0,08
Próx. direção	0,05	0,09
Próx. residual	0,05	0,09
Próx. tentativa	0,05	0,09
Matriz vetor	0,17	0,28
Sobrecarga	0,24	0,26
Tempo total	0,67	0,91

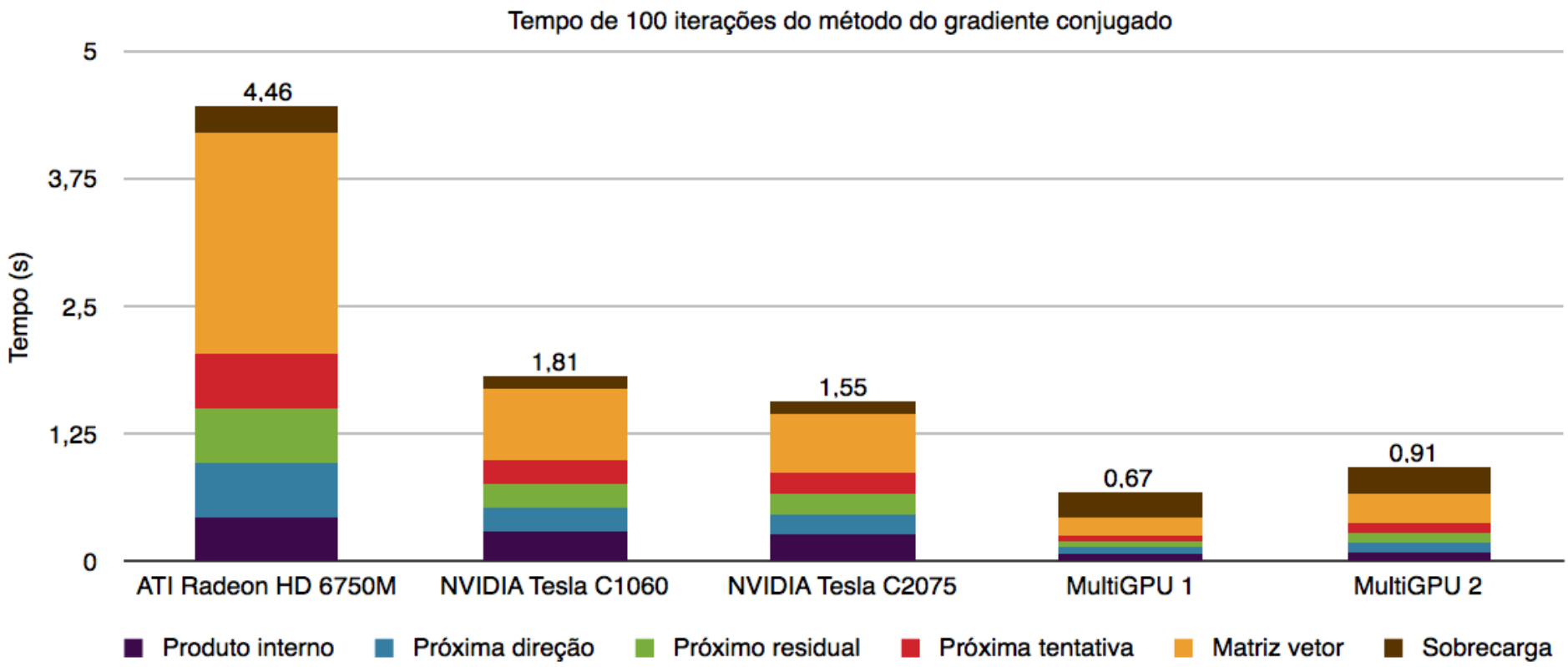

Figura 4.21: Tempo de execução de 100 iterações do método do gradiente conjugado para uma matriz de ordem 10^7 em diferentes plataformas.

Adicionalmente calculamos o número de operações de ponto flutuante por segundo (FLOP/s) atingido por cada dispositivo. Essa é uma boa métrica usada para determinar o desempenho de uma aplicação. Já que as GPUs possuem enorme capacidade de processamento, convém utilizar unidades maiores que FLOPs, como o múltiplo gigaFLOP/s (GFLOP/s).

O primeiro passo é calcular o número de GFLOP/s teórico, isto é, o máximo de desempenho que a placa pode oferecer na execução de determinado kernel. Esse valor é calculado através da seguinte expressão:

$$GFLOP/s\ teórico = \frac{FLOP}{Bytes\ lidos\ da\ memória\ global} * largura\ de\ banda$$

Dividimos o número de FLOPs efetuado para cada conjunto de Bytes lidos da memória global e multiplicamos o resultado pela largura de banda de memória da GPU. As larguras de banda das GPUs utilizadas são:

. ATI Radeon HD 6750M - 57,6 GB/s.

. NVIDIA Tesla C1060 - 102 GB/s.

. NVIDIA Tesla C2075 - 148 GB/s.

Para o cálculo do número de GFLOP/s prático efetuamos o cálculo:

$$GFLOP/s\ prático = \frac{FLOP * saltos * número\ de\ itens\ de\ trabalho}{tempo}$$

Multiplicamos o número de FLOPs efetuado por todos os itens de trabalho. Para isso devemos multiplicar pelo número de saltos que cada item de trabalho dá para percorrer toda a estrutura de dados que está sendo lida. Dividimos o resultado pelo tempo de execução de uma iteração do kernel.

As tabelas 4.7, 4.9 e 4.11 apresentam os resultados teóricos do cálculo de GLFLOP/s. As tabelas 4.8, 4.10 e 4.12 os resultados obtidos nos testes e também a porcentagem alcançada em relação ao resultado teórico. Finalmente a figura 4.22 sumariza esses resultados através de uma representação gráfica.

Tabela 4.7: Cálculo de GFLOP/s teórico — ATI Radeon HD 6750M

Kernel	Leitura mem. global (Bytes)	FLOP	GFLOP/s teórico
Prod. interno	8	1	7,2
Próx. dir./res/tent.	12	2	9,6
Matriz vetor	44	2	2,6

Tabela 4.8: Cálculo de GFLOP/s prático — ATI Radeon HD 6750M

Kernel	tempo (s)/ iteração	GFLOP/s prático	% teórico
Prod. interno	0,0022	6,9	95%
Próx. dir./res/tent.	0,0053	5,8	60%
Matriz vetor	0,0216	1,4	53%

Tabela 4.9: Cálculo de GFLOP/s teórico — NVIDIA Tesla C1060

Kernel	Leitura mem. global (Bytes)	FLOP	GFLOP/s teórico
Prod. interno	8	1	12,7
Próx. dir./res/tent.	12	2	17,0
Matriz vetor	44	2	8,5

Tabela 4.10: Cálculo de GFLOP/s prático — NVIDIA Tesla C1060

Kernel	tempo (s)/ iteração	GFLOP/s prático	% teórico
Prod. interno	0,0015	10,2	80%
Próx. dir./res/tent.	0,0023	13,4	78%
Matriz vetor	0,0069	4,4	51%

Tabela 4.11: Cálculo de GFLOP/s teórico — NVIDIA Tesla C2075

Kernel	Leitura mem. global (Bytes)	FLOP	GFLOP/s teórico
Prod. interno	8	1	18,5
Próx. dir./res/tent.	12	2	24,6
Matriz vetor	44	2	12,3

Tabela 4.12: Cálculo de GFLOP/s prático — NVIDIA Tesla C2075

Kernel	tempo (s)/ iteração	GFLOP/s prático	% teórico
Prod. interno	0,0013	11,8	64%
Próx. dir./res/tent.	0,0020	15,3	62%
Matriz vetor	0,0057	5,4	44%

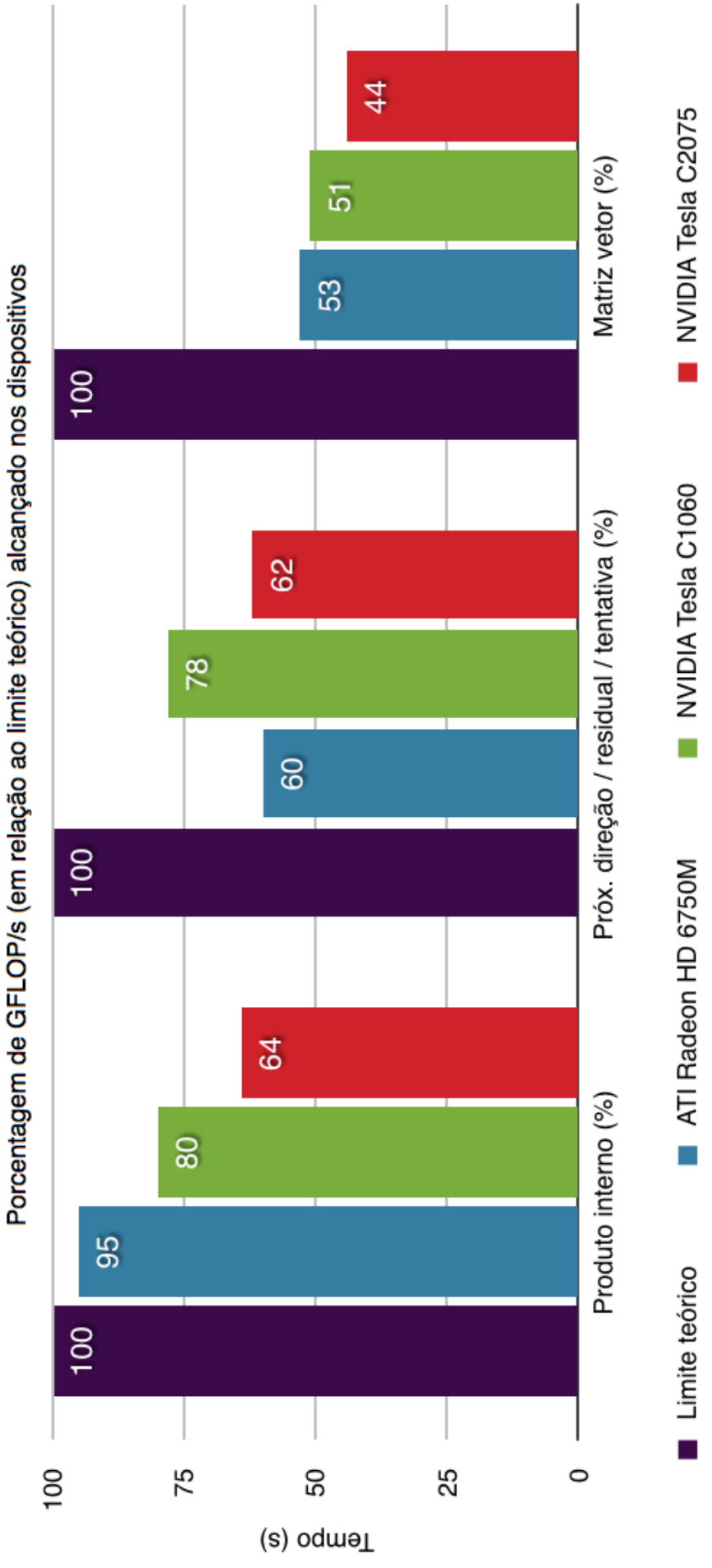

Figura 4.22: Porcentagem de GFLOP/s (em relação ao limite teórico) alcançado nos dispositivos.

5
Conclusão

Esse trabalho apresentou uma forma de acelerar a resolução de sistemas de equações lineares de grande porte utilizando múltiplas GPUs. A proposta foi validada com experimentos que usaram matrizes da coleção Harwell-Boeing NIST [10]. Os kernels, implementados de forma modular, podem ser reutilizados em outros métodos numéricos, garantindo assim o reaproveitamento da implementação.

Com um maior entendimento da linguagem OpenCL observamos a necessidade de expandir a motivação inicial do trabalho, que era apenas a resolução dos sistemas, para também englobar a automatização da escolha do número de grupos de trabalhos por unidade de computação. Essa foi uma necessidade natural advinda da possibilidade de execução desse sistema em qualquer dispositivo GPU que suporte OpenCL, já que as características inerentes de cada hardware levam a necessidades diferentes de carga de trabalho. Descobrimos que essa automatização, além de facilitar métodos de tentativa e erro, gera um ganho significativo de tempo a cada execução do método. Tentamos encontrar, sem sucesso, uma fórmula geral de divisão de trabalho em um conjunto arbitrário de dispositivos, porém descobrimos que esse insucesso foge do controle do programador, já que a linguagem OpenCL não garante a paralelização de grupos de trabalho, fazendo com que certos dispositivos teoricamente mais poderosos sejam subutilizados.

Finalmente mostramos que a abordagem de resolução utilizando múltiplas GPUs é vantajosa. Apesar da sobrecarga advinda do uso de múltiplos dispositivos, obtivemos uma redução de tempo de 63% na plataforma MultiGPU 1 (três placas NVIDIA Tesla C1060) quando comparado com apenas uma placa do mesmo tipo e uma redução de tempo de 41% na plataforma MultiGPU 2 (duas placas NVIDIA Tesla C2075) quando comparado com apenas uma placa do mesmo tipo.

Sugerimos uma expansão do sistema que suporte outras formas de armazenamento da matriz como um trabalho futuro, bem como a integração kernels, em OpenCL, otimizados especificamente para CPUs.

6
Referências Bibliográficas

[1] C. Ashcraft, R. Grimes, J. Lewis, B. Peyton, and H. Simon. Progress in sparse matrix methods for large sparse linear systems on vector supercomputers. *Intern. J. of Supercomputer Applications*, 1:10,30, 1987.

[2] M. Arioli, J. W. Demmel, and I.S. Duff. Solving sparse linear systems with sparse backward error. SIAM J. Matrix Anal. Appl., 10(2):165-190, April 1989.

[3] I.S. Duff and J.K. Reid. MA48, a Fortran code for direct solution of sparse unsymmetric linear systems of equations. Technical Report RAL-93-072, Rutherford Appleton Laboratory, Oxon, UK, 1993.

[4] I.S. Duff and J.K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. ACM Trans. Mathematical Software, 9:302-325, 1983.

[5] A.H. Sherman. On the eficient solution of sparse systems of linear and nonlinear equations. PhD thesis, Yale University, 1975.

[6] T.A. Davis and I. S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. Technical Report RAL 93-036, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, 1994.

[7] J.R. Shewchuk, *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*, Edition 1 $\frac{1}{4}$, August 4, 1994.

[8] OpenCL, The open standard for parallel programming of heterogeneous systems, `http://www.khronos.org/opencl`

[9] Owens, John D.; Houston, Mike; Luebke, David; Green, Simon; Stone, John E.; Phillips, James C. , *GPU Computing*, 2008.

[10] Matrix Market,
`http://math.nist.gov/MatrixMarket`, May 10, 2007.

[11] `http://math.nist.gov/MatrixMarket/mmio-c.html`

[12] Developer, AMD,
`http://developer.amd.com/documentation/articles/Pages/`
`OpenCL-Optimization-Case-Study-Simple-Reductions.aspx`

[13] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, J. Demmel, Optimization of sparse matrix-vector multiplication on emerging multicore platforms, in: SC '07: Proc. of the 2007 ACM/IEEE conf. on Supercomputing, pp. 38:1–38:12.

[14] W. Wiggers, V. Bakker, A. Kokkeler, G. Smit, Implementing the conjugate gradient algorithm on multi-core systems, in: Proc. of the Int. Symp. on System-on-Chip (SoC 2007), pp. 11–14.

[15] J. Bolz, I. Farmer, E. Grinspun, P. Schroder, Sparse matrix solvers on the GPU: Conjugate gradients and multigrid, in: Proc. SIGGRAPH'03, pp. 917–924.

[16] L. Buatois, G. Caumon, B. L'evy, Concurrent Number Cruncher : An Efficient Sparse Linear Solver on the GPU, in: High Performance Computation Conference - HPCC'07, pp. 358–371.

[17] N. Bell, M. Garland, Efficient Sparse Matrix-Vector Multiplication on CUDA, Technical Report NVR-2008-004, NVidia, 2008.

[18] A. Monakov, A. Avetisyan, Implementing Blocked Sparse Matrix-Vector Multiplication on NVIDIA GPUs, in: SAMOS '09: Proc. of the 9th Int. Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation, pp. 289–297.

[19] J.W. Choi, A. Singh, R.W. Vuduc, Model-driven autotuning of sparse matrix-vector multiply on GPUs, SIGPLAN Not. 45 (2010) 115–126.

[20] A. Cevahir, A. Nukada, S. Matsuoka, Fast Conjugate Gradients with Multiple GPUs, in: Computational Science ICCS 2009, pp. 893–903.

[21] A. Cevahir, A. Nukada, S. Matsuoka, High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning, Computer Science - Research and Development 25 (2010) 83–91.

[22] `http://viennacl.sourceforge.net`

[23] `http://ixbtlabs.com/articles3/video/cuda-1-p1.html`

[24] `http://www.cmsoft.com.br/index.php ?option=com_contentview=category&layout=blog&id=60&Itemid=97`

[25] `http://viennacl.sourceforge.net/viennacl-benchmarks.html`

[26] Adve, Sarita V., and Kourosh Gharachorloo. "Shared memory consistency models: A tutorial."computer 29.12 (1996): 66-76.

[27] Munshi, Aaftab, Benedict Gaster, and Timothy G. Mattson. OpenCL programming guide. Addison-Wesley Professional, 2011

[28] http://www-igm.univ-mlv.fr/~ dr/XPOSE2008/CUDA_GPGPU/optimisations.html

[29] NVIDIA OpenCL Programming Overview

[30] http://pubs.opengroup.org/onlinepubs/007904975/basedefs/stdint.h.html

[31] http://www.nersc.gov/users/computational-systems/dirac/performance-and-optimization/

[32] http://netlib.org/blas/