UNIVERSIDADE ESTADUAL DE LONDRINA

RICARDO INÁCIO ÁLVARES E SILVA

DECOMPOSIÇÃO DE MULTIPLICAÇÃO MATRICIAL EM TAREFAS ADEQUADAS A AMBIENTES HETEROGÊNEOS BASEADOS EM GPGPU

Londrina
2013

Universidade Estadual de Londrina

RICARDO INÁCIO ÁLVARES E SILVA

MESTRADO - 2013

DC
Departamento de Computação

Programa de Mestrado em Ciência da Computação

Universidade Estadual de Londrina

DC
Departamento de Computação

Universidade Estadual de Londrina

Rodovia Celso Garcia Cid - PR 445 Km 380

Campus Universitário - Universidade Estadual de Londrina

Catalogação elaborada pela Divisão de Processos Técnicos da Biblioteca Central da Universidade Estadual de Londrina

Dados Internacionais de Catalogação-na-Publicação (CIP)

S586d Silva, Ricardo Inácio Álvares e.
Decomposição de multiplicação matricial em tarefas adequadas a ambientes heterogêneos baseados em GPGPU / Ricardo Inácio Álvares e Silva. – Londrina, 2013.
76 f. : il. + anexo no final da obra.

Orientador: Jacques Duílio Brancher.
Dissertação (Mestrado em Ciência da Computação) – Universidade Estadual de Londrina, Centro de Ciências Exatas, Programa de Pós-Graduação em Ciência da Computação, 2013.
Inclui bibliografia.

1. Programação paralela (Computação) – Teses. 2. Processamento paralelo (Computadores) – Teses. 3. Álgebra linear – Processamento de dados – Teses. 4. Sistemas operacionais distribuídos (Computadores) – Teses. I. Brancher, Jacques Duílio. II. Universidade Estadual de Londrina. Centro de Ciências Exatas. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDU 519.684

RICARDO INÁCIO ÁLVARES E SILVA

DECOMPOSIÇÃO DE MULTIPLICAÇÃO MATRICIAL EM TAREFAS ADEQUADAS A AMBIENTES HETEROGÊNEOS BASEADOS EM GPGPU

Trabalho de Dissertação de Mestrado apresentado à Universidade Estadual de Londrina como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Jacques Duílio Brancher.

Londrina
2013

RICARDO INÁCIO ÁLVARES E SILVA

DECOMPOSIÇÃO DE MULTIPLICAÇÃO MATRICIAL EM TAREFAS ADEQUADAS A AMBIENTES HETEROGÊNEOS BASEADOS EM GPGPU

Trabalho de Dissertação de Mestrado apresentado à Universidade Estadual de Londrina como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.

BANCA EXAMINADORA

_____
Prof. Dr. Jacques Duílio Brancher
Universidade Estadual de Londrina – UEL

_____
Prof. Dr. Bruno Bogaz Zarpelão
Universidade Estadual de Londrina - UEL

_____
Prof. Dr. Wesley Attrot
Universidade Estadual de Londrina - UEL

_____
Profa. Dra. Neyva Maria Lopes Romeiro
Universidade Estadual de Londrina – UEL

Londrina, 20 de fevereiro de 2013.

Este trabalho é dedicado às pessoas que
mais amo. Meus pais pela vida.
Meu irmão pela companhia.
Minha mulher pela aventura.
Meus amigos pela fidelidade.

AGRADECIMENTOS

Ao Professor Jacques Brancher pela paciência, amizade e principalmente pelas oportunidades a mim oferecidas, longe de se resumirem apenas à chance de ingressar no programa de mestrado. Aos Professores Wesley, Alan e Rodolfo pela aulas ministradas, conselhos e companhia. Às secretárias Rosana e Valdete pela presteza e o bom humor de sempre.

Aos colegas de laboratório e novos amigos, Thiago Sotana e Marc, pioneiros na aventura, que muito me ajudaram com boa vontade, paciência, conhecimento e boas conversas. Sem eles teria sido impossível encarar métodos numéricos e álgebra linear. Aos calouros da 310, Estevan e Fábio, sempre com disposição para ajudar e participativos. Aos colegas de mestrado em geral pela ótima convivência em sala de aula.

À cidade de Londrina e à UEL, que se mostraram receptivas e me permitiu fazer novos e valiosos amigos. Ao pessoal do laboratório da Professora Eiko, do CCB, pelo apoio e por aturar minhas interrupções em seus trabalhos para pegar a chave do carro. Agradeço à Luciene, Fabiana, Juliana, Maria Cláudia, Nilson, Mary e demais.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pela valiosa assistência através de bolsa de estudos, sem a qual seria impossível minha dedicação em tempo integral aos estudos e produção científica.

Um agradecimento especial ao Professor Lucas Ferrari de Oliveira, da Universidade Federal do Paraná, pelo livre acesso remoto, via *ssh*, a seu *workstation*, com direitos de *root* e tudo! Eu não acredito que o trabalho teria chegado a este ponto sem essa enorme contribuição e demonstração de confiança no próximo. Espero poder retribuir no futuro próximo.

“Well, I believe in what you say,
It’s the undisputed truth.
But I have to have things my own way,
To keep me in my youth.”
– *Goodbye Stranger*, Supertramp

SILVA, Ricardo Inácio Álvares e. **Decomposição de Multiplicação Matricial em Tarefas Adequadas a Ambientes Heterogêneos Baseados em GPGPU**. 2013. nnn f. Dissertação (Mestrado) - Universidade Estadual de Londrina, Londrina. 2013.

RESUMO

Este trabalho desenvolve dois algoritmos para decomposição de multiplicação matricial geral (GEMM, do inglês *General Matrix Multiplication*) em tarefas menores, adequadas à distribuição entre processadores disponíveis em sistemas heterogêneos, como CPUs e GPUs, que têm como objetivo a escalabilidade de desempenho. Um dos algoritmos gera tarefas que são multiplicações matriciais menores, independentes entre si, mas com redundância de dados nas transferências entre os processadores. O trabalho mostra que esse algoritmo também pode ser utilizado para explorar recursos de GPUs como a sobreposição de operações de transferências e execução de *kernels*. Já o outro algoritmo decompõe multiplicações matriciais em três grupos de tarefas, sendo dois de multiplicações menores e um de somas vetoriais. Demonstra-se que apesar das tarefas possuírem independência apenas parcial, podem ser organizadas de tal forma a serem resolvidas concorrentemente. Este algoritmo também prevê a viabilidade de redução nas transferências de memória entre os processadores e evita redundância nas transferências de dados. O trabalho ainda mostra que tais características são desejáveis para sistemas computacionais heterogêneos baseados em computação de propósito geral na unidade de processamento gráfico (GPGPU, do inglês *General Purpose computing on GPU*). Por fim, o trabalho apresenta implementações dos algoritmos propostos e seus respectivos desempenhos. A experimentação mostra que, em um sistema com duas GPUs, os algoritmos podem melhorar o desempenho de multiplicações em 50%, para matrizes de dimensão 1000×1000, até acima de 100%, para 4000×4000 adiante.

Palavras-Chave: Álgebra Linear, Basic Linear Algebra Library (BLAS), Sistemas Heterogêneos, Paralelismo, Escalabilidade, Desempenho.

SILVA, Ricardo Inácio Álvares e. **Decomposition of Matrix Multiplication into Task for GPGPU-based Heterogeneous Environments**. 2013. nnn f. Dissertation (Master's Degree) - State University of Londrina, Londrina. 2013.

ABSTRACT

In this work, we developed two algorithms for decomposing a general matrix multiplication (GEMM) into smaller tasks, adequate to distribution between available processors in heterogeneous systems, such as CPUs and GPUs, that have performance scalability as its main objective. One of the algorithms generates independent smaller matrix multiplication tasks, but with redundancy in data transfers between processors. We show that this algorithm can utilize GPUs features such as overlapping of memory transfers and kernel executions. The other algorithm presented decomposes matrix multiplications in three groups of tasks, two being smaller multiplications and, the other, vectorial sums. We show that although the independence of these tasks being only partial, they can be organized in such a way that are able to be solved concurrently. This algorithm also provides reduction in memory transfers between processors and avoids redundancy. We also show that such features are desirable in heterogeneous systems based in general purpose computing in GPU (GPGPU). Finally, we show implementations to the proposed algorithms and their respective performance. Experimentation shows that, in a system with two GPUs, they can improve matrix multiplication performance by 50%, for matrix dimensions from 1000×1000, up to over 100%, for 4000×4000 and on.

Keywords: Linear Algebra, Basic Linear Algebra Library (BLAS), Heterogeneous Environments, GPU, Parallelism, Scalability, Performance.

Lista de ilustrações

Lista de tabelas

Lista de Algoritmos

LISTA DE SIGLAS E ABREVIATURAS

API	*Application programming interface*
AXPY	*Alpha X Plus Y*
CPU	*Central Processing Unit*
DGEMM	*Double precision GEMM*
DMA	*Direct Memory Access*
E/S	Entrada e Saída
EC2	*Elastic Compute Cloud*
FFT	*Fast Fourier Transform*
FLOPS	*Floating-point Operations per Second*
FPGA	*Field-Programmable Gate Array*
GB/s	*Gigabytes* por segundo
GEMM	*General Matrix Multiplication*
GEMV	*General Matrix-Vector Multiplication*
GEPM	*General Panel-Matrix Multiplication*
GEPP	*General Panel-Panel Multiplication*
GFLOPS	Bilhões de FLOPS
GPGPU	*General Purpose computation on GPU*
GPU	*Graphics Processor Unit*
LU	*Lower-Upper*
MB	*Megabyte*
SO	Sistema Operacional
PCIe	*Peripheral Component Interconnect Express*
PD	Precisão Dupla

PS Precisão Simples

RAM *Read-Only Memory*

SAXPY *Single precision AXPY*

SGEMM *Single precision GEMM*

SIMT *Single Instruction Multiple Threads*

SMP *Symmetric Multiprocessor*

TFLOPS Trilhões de FLOPS

TLB *Translation Lookaside Buffer*

UFPR Universidade Federal do Paraná

VRAM *Video RAM*

Sumário

1 INTRODUÇÃO

Os computadores surgiram, em princípio, para resolver a grande quantidade de cálculos necessários para a resolução de problemas modernos da matemática e da física. Com eles, foi possível melhorar a precisão de cálculos, realizar simulações e se aprofundar em novos e complexos problemas. No entanto, à medida que tais problemas se desenvolviam, aumentavam a quantidade de cálculos a serem feitos. Portanto, era necessário melhorar o desempenho computacional para que a máquina retornasse os resultados no menor tempo possível.

No esforço para ganho de desempenho, duas abordagens foram tomadas. Uma delas é a melhora da tecnologia dos processadores, na forma de arquiteturas mais eficientes e também na melhora do processo de produção, principalmente pela miniaturização. Dessa forma, uma rotina matemática ganhava desempenho sem que tivesse seu código alterado. A outra abordagem é a da adequação dessas rotinas para a arquitetura específica do processador que se encarregaria do trabalho. Esse processo, também conhecido como otimização, envolve a busca pela utilização máxima dos recursos computacionais disponíveis.

Contudo, ultimamente, a taxa de aumento do desempenho das tradicionais *Central Processing Unit*s (CPUs) a cada nova geração têm sido aquém do nível histórico, abaixo do esperado [1]. Como cientistas e desenvolvedores de sistemas se acostumaram com o desenvolvimento da forma que era, inclusive com a definição da informal Lei de Moore, o andamento e aprofundamento de diversos projetos correm risco de diminuir de ritmo conjuntamente. Por isso novas abordagens têm sido tentadas, como o aumento de núcleos de CPU em um único *chip* e a introdução de aceleradores, como *Field-Programmable Gate Array* (FPGA) e *Graphics Processor Unit* (GPU), aos sistemas tradicionais [2].

Neste sentido, as GPUs surgiram para resolver a necessidade de renderização em aplicações de computação gráfica e jogos eletrônicos. A tarefa de gerar uma imagem de duas dimensões a partir da descrição de um modelo de três dimensões envolve uma grande quantidade de cálculos, semelhantes para cada ponto da tela a ser colorido. Por esse motivo, foram desenvolvidos *chips* dedicados capazes de realizar tais cálculos com alto grau de paralelismo. À medida em que evoluíam, foram se tornando flexíveis até se tornarem completamente programáveis, através da incorporação de núcleos *shader*, que permitiam a modificação dos programas *shaders*, responsável por transformações de cor, brilho e outras propriedades dos *pixels*, a serem utilizados na renderização de modelos em três dimensões.

Desde então, pesquisadores têm avaliado a possibilidade de utilizar as GPUs para outras tarefas que não a original. Sua utilização nessas novas tarefas é conhecida por *General Purpose computation on GPU* (GPGPU). Devido à arquitetura de elevado nível de paralelismo no processamento, alguns importantes algoritmos utilizados no meio científico,

como *Fast Fourier Transform* (FFT) e operações matriciais, mostraram resultados promissores.

Os primeiros trabalhos com essa tecnologia foram desenvolvidos distorcendo o modelo de programação existente, orientado para processamento de imagens. Os programadores tinham que mapear os dados relevantes de seu problema em um objeto na memória originalmente projetado para descrever texturas, e desenvolver o algoritmo como um programa *shader*. Por este ser um método não natural, sujeito a erros, imprecisões e que dificulta a abstração, linguagens e ferramentas específicas para programação de propósito geral em GPU foram desenvolvidas. Dentre elas estão Cg, Brook+, CUDA [3] e OpenCL [4], sendo que as duas últimas encaminham para se tornarem os modelos padrões, em vista da quantidade de artigos científicos sobre elas ou que as utilizam.

Além disso, pesquisas apontam para a possibilidade de utilização de GPGPU como um núcleo específico, acelerador da CPU, concorrente com seus próprios núcleos de propósito geral [5, 6, 7]. Um sistema com essa configuração é denominado como um ambiente heterogêneo, devido a utilização de núcleos de diferentes natureza para uma mesma tarefa.

O desempenho é essencial para o desenvolvimento de aplicações dependentes de modelos matemáticos, e dentre estes, destaca-se a álgebra linear. Ela é fundamental para diversos fins científicos, especialmente os relacionados à física e simulação de modelos teóricos. Em seu domínio, identifica-se a multiplicação matricial como uma importante operação, pois é necessária a diversos algoritmos. Além disso, por ser uma operação cara em termos computacionais, tem-se que uma implementação otimizada é vital para o desempenho de outras operações dependentes dela.

Por esse motivo, a LAPACK [8], uma biblioteca de funções de álgebra linear, foi desenvolvida sobre uma outra biblioteca, a BLAS [9], que implementa operações vetoriais e matriciais de maneira otimizada para o desempenho [10]. Uma delas é a multiplicação matricial, denominada *General Matrix Multiplication* (GEMM). Essa abordagem se mostrou acertada a ponto de ambas bibliotecas se tornarem referências na melhora do desempenho relacionado a área. Como consequência, os nomes das rotinas e suas interfaces de chamada são tomadas como um padrão, e são utilizadas nas implementações de várias outras bibliotecas relevantes de funções de álgebra linear, como a MKL [11], ATLAS [12], gotoBLAS [13], CUBLAS [14], FLAME [15] e ACML [16].

1.1 O PROJETO

Como ponto de partida, foi identificada a possibilidade de evolução em projeto e execução de algoritmos para multiplicação matricial adequados a ambientes heterogêneos. No momento em que esse tipo de sistema computacional se apresenta como uma possibilidade importante de futuro da evolução do desempenho, nenhuma das tradicionais bibliotecas apresentam uma solução que busque a utilização dos diversos processadores disponíveis para a execução de GEMM de maneira ótima.

Portanto, o objetivo do trabalho encontra-se no desenvolvimento de algoritmos que realizem a decomposição de uma operação GEMM em um conjunto de tarefas independentes entre si. Dessa forma, a operação como um todo poderia ser paralelizada e executada concorrementemente por processadores distintos, disponíveis em um sistema. Como nesse tipo de sistema as operações de transferência de memória entre os processadores são fator limitante, é interessante que novas rotinas consigam evitá-las ao máximo.

O que se espera de novas rotinas é competitividade em relação as das melhores bibliotecas existentes, do ponto de vista do desempenho computacional, para casos comuns de execução em sistemas com ao menos uma GPU e uma CPU, especificamente para matrizes densas e de tamanhos elevados. E espera-se que elas sobressaiam e sejam vantajosas em sistemas com grande quantidade de aceleradores GPUs, mostrando capacidade para escalabilidade de desempenho à medida em que se adiciona mais poder de computação na forma de adição de novos processadores ao sistema.

Este trabalho espera dar uma contribuição a este ramo científico da computação, busca construir bases necessárias para o desenvolvimento de bibliotecas de métodos matriciais adequados a ambientes heterogêneos, com múltiplos processadores, distintos, distribuídos e concorrentes. Nesse tipo de sistema, não bastam grandes problemas a serem resolvidos, mas maneiras de subdividi-los em partes menores e também classificar essas partes menores quanto a afinidade operacional, de forma que um escalonador inteligente possa fazer combinações de agendamento de execução de tarefas razoáveis.

Outra contribuição esperada, comum a todos os trabalhos de otimização de GEMM em sistemas computacionais, é a clarificação sobre os pontos de retenção dos atuais computadores. Ao mesmo tempo em que se desenvolve algoritmos com vistas a minimizar o efeito de uma deficiência arquitetural do sistema sobre o desempenho, o trabalho pode mostrar aos projetistas qual caminho seguir na melhora do estado da arte da arquitetura de computadores.

Para este fim, o trabalho está organizado da seguinte forma: no capítulo 2 são apresentados trabalhos científicos relacionados ao desenvolvimento de algoritmos de multiplicação matricial para computadores; no capítulo 3 são abordados detalhes relevantes para a modelagem e programação de algoritmos em sistemas heterogêneos, que foram considerados e utilizados na implementação da solução proposta por este trabalho; no capítulo 4 são desenvolvidos algoritmos para execução de multiplicação matricial em ambientes heterogêneos de tal forma que seja possível utilizar todos os processadores disponíveis e minimizar as transferências de memória; no capítulo 5 é discutida a implementação e experimentação do algoritmo proposto; por fim, no capítulo 7 é feita a conclusão do trabalho e considerações sobre propostas para trabalhos futuros.

2 MÉTODOS DE MULTIPLICAÇÃO MATRICIAL

O capítulo está organizado da seguinte forma: a seção 2.1 apresenta a relação entre multiplicação matricial e arquitetura de computadores na evolução do desempenho; a seção 2.2 apresenta trabalhos correlatos.

2.1 MULTIPLICAÇÃO MATRICIAL EM COMPUTADORES

De acordo com Dongarra [17], a engenharia de *softwares* matemáticos para álgebra linear como se conhece hoje começou no início da década de 1970 com o EISPACK [18]. Os principais objetivos da biblioteca eram a portabilidade, facilidade de uso e confiabilidade dos resultados numéricos. Apesar das limitações tecnológicas da época, através de comunicação e distribuição *ad hoc*, ela foi testada em vários sistemas. Por esse motivo, foi bem recebida e se tornou conhecida.

Ainda na narrativa de Dongarra [17], com os problemas anteriores resolvidos e a evolução e padronização da representação de números de ponto flutuante, iniciou-se o desenvolvimento da LINPACK [19]. Com ela, buscou-se a disponibilização de novas funções e, principalmente, a sua adequação a ambientes de super computadores, ou *main-frames*. A ideia era atingir melhor desempenho ao explorar as características das arquiteturas dos sistemas, tanto é que o manual de uso da biblioteca vinha com tabelas como referência indicando tempos de execução de suas rotinas na máquina do desenvolvedor. Isso desencadeou uma competição de supercomputadores, a *Top 500 Supercomputers* [20], e definiu a tendência no desenvolvimento desse tipo de *software*, a busca por maior desempenho a cada nova arquitetura desenvolvida.

2.1.1 Pacotes de núcleos e GEMM

Com o surgimento dos microprocessadores (microcomputadores) e com a melhora da própria engenharia de *software*, verificou-se a necessidade de separar as rotinas matriciais em vários níveis. Dongarra [17] explica que surgia uma grande quantidade de arquiteturas de sistemas, e o trabalho para adaptar cada rotina a cada uma delas era proibitivo. Portanto, com a separação em níveis seria possível identificar as rotinas mais utilizadas e concentrar esforços de otimização.

Desta forma, foram identificadas operações básicas, denominadas núcleos (do inglês *kernels*), que eram muito utilizadas e as funções foram separadas nas novas bibliotecas LAPACK e BLAS [8, 9]. A primeira implementa as operações vetoriais que são utilizadas pela segunda na implementação de algoritmos de álgebra linear. Adiante, identificaram novos núcleos, que resultou na criação da BLAS nível 2, que implementa operações entre matrizes e vetores, e nível 3, que implementa as puramente matriciais.

Dentre os núcleos mais relevantes estão a soma entre dois vetores e a multiplicação matricial geral. O primeiro é denominado *Alpha X Plus Y* (AXPY), pertence à BLAS nível 1 e é definido por

$$y \leftarrow \alpha \cdot x + y,$$

em que $x, y \in \mathbb{R}^n$, com $n \in \mathbb{N}$, são os vetores a serem adicionados. Por ser comum em processadores a existência de instruções que fazem multiplicação seguida de soma em um único ciclo, a operação aceita o parâmetro $\alpha \in \mathbb{R}$ que multiplica x, o que provê possibilidades de otimização. Nota-se a utilização do símbolo $\leftarrow$ (seta para esquerda), que indica que o resultado será guardado na posição de memória de y, sobrescrevendo os valores anteriores.

Já a multiplicação matricial geral é denominada GEMM, pertence à BLAS nível 3 e é definida por

$$C \leftarrow \alpha \cdot A^{t_a} B^{t_b} + \beta \cdot C,$$

em que $A \in \mathbb{R}^{n \times k}$ e $B \in \mathbb{R}^{k \times m}$ são as matrizes a serem multiplicadas, com n, m e $k \in \mathbb{N}$, e os parâmetros t_a e t_b indicam a ordem de transposição dos dados de A e B, respectivamente, na memória, ou seja, indicam organização em vetores colunas ou em vetores linhas. O resultado de $A^{t_a} B^{t_b}$ é uma matriz $(AB) \in \mathbb{R}^{n \times m}$ cujos elementos são definidos por

$$(AB)_{i,j} = \sum_{k=1}^{p} A_{i,k} B_{k,j}.$$

(AB) é multiplicada por um escalar $\alpha \in \mathbb{R}$ e em seguida somada a $C \in \mathbb{R}^{n \times m}$, que antes é multiplicada por um escalar $\beta \in \mathbb{R}$. O resultado de toda a operação é retornado em C. A função das multiplicações das matrizes pelos valores escalares é permitir a acumulação de resultados sem necessitar de operações extra de memória para duplicar alguma das matrizes [10].

Um detalhe relevante em programação de métodos numéricos é o tipo dos dados a serem processados, se são reais ou complexos e se são de precisão simples ou dupla. BLAS possui um núcleo de cada função específico para cada um desses tipos de dados, que são indicados por um prefixo em seu nome, de acordo com a Tabela 2.1. Por exemplo, o núcleo GEMM para números reais de dupla precisão é denominado *Double precision GEMM* (DGEMM), enquanto AXPY para precisão simples é *Single precision AXPY* (SAXPY).

Tabela 2.1 – Significado dos prefixos de nomenclatura dos núcleos BLAS.

Prefixo	Significado
S	Números reais de precisão simples.
D	Números reais de precisão dupla.
C	Números complexos de precisão simples.
Z	Números complexos de precisão dupla.

Fonte: Geijn e Quintana–Ortí [21].

Após essa separação em níveis, o núcleo GEMM se tornou o mais importante dos que compõem a BLAS, pois é o mais caro computacionalmente com custo assintótico de

$O(n^3)$, para o algoritmo tradicional [10]. Além disso, é nele que se perde mais tempo nos vários algoritmos da LAPACK em que está presente. Por esses motivos, se tornou essencial que sua implementação tivesse desempenho o mais próximo do ótimo possível [22]. Por fim, GEMM é uma rotina cujo desempenho depende tanto da qualidade geral da memória quanto da capacidade computacional das unidades lógicas aritméticas dos processadores. Portanto, é uma rotina utilizada para caracterizar as qualidades de um sistema computacional, e indicar aos projetistas os pontos de contenção ao desempenho ótimo do sistema.

2.1.2 Modernização de GEMM para CPUs

Voltando a Dongarra [17], inicialmente a busca pelo GEMM ótimo era encontrar uma combinação de instruções que fosse rápida, por causa da simplicidade dos microprocessadores da época. Mais adiante, surgiram os processadores vetoriais que causaram o início da análise da organização das matrizes na memória. Isso porque, para que fossem usados eficientemente, os registradores vetoriais precisavam ser carregados rapidamente.

Em seguida, surgiu a hierarquia de memórias em microprocessadores populares, em que a combinação de instruções já não era o fator mais relevante ao desempenho. A busca passou a ser a da organização do problema em estruturas de dados que coubessem em cada nível de memória de tal forma a manter o processador abastecido com instruções e dados pelo maior tempo possível [23]. Em outras palavras, quando um conjunto de dados é transferido da memória mais lenta para alguma memória de melhor desempenho, tais dados devem permanecer por lá até que sejam usados em todos os casos necessários. Nem sempre é possível atingir essa condição ótima, mas a ideia é maximizar o reuso [22].

De acordo com Whaley e Dongarra [24], o manuseio da hierarquia de memórias nos processadores acontece de forma implícita e não exata. Essa característica dificulta a busca pelo método com desempenho ótimo, apesar de possível. Porém, ao considerar que existem vários processadores com arquiteturas distintas, e que mesmo entre os com arquitetura similar a configuração das memórias muda, verificou-se a falta de portabilidade dos códigos. A solução encontrada para o problema foi a utilização de *auto-tuning*, ou técnicas de otimização automática, que consiste em executar vários testes na máquina no momento da instalação da biblioteca para extrair parâmetros que gerem a melhor implementação. Com isso foi criado o *software* ATLAS, que exerce essa função sobre a LAPACK. Outros trabalhos também utilizam técnicas de otimização automática para CPUs com sucesso, como em D'Alberto e Nicolau [25], e Rünger e Schwind [26].

A multiplicação matricial é uma operação bastante flexível, que pode ser resolvida de maneiras distintas, inclusive com a sua decomposição em operações menores [21, 22]. Dentre as possibilidades estão a decomposição em operações de multiplicação, operações de multiplicação matriz-vetor e também operações de multiplicação puramente vetoriais. Cada uma dessas operações possui um núcleo ou subnúcleo BLAS.

A Tabela 2.2 ilustra os possíveis núcleos, dados os tamanhos m e n para a

matriz resultante, e k como a dimensão comum para as matrizes de entrada. A última coluna indica o nome dado ao núcleo, que segue uma lógica de sufixos baseados nas dimensões das matrizes de entrada e explicada pela Tabela 2.3. Por exemplo, se a multiplicação é entre matriz de dimensões grandes e vetor, o nome do núcleo é *General Matrix-Vector Multiplication* (GEMV). A exceção é o caso GEPDOT, sem abreviatura específica, que, assim como *General Panel-Panel Multiplication* (GEPP), multiplica duas matrizes painéis, porém em ordem invertida, e recebe outro nome pois é operacionalmente distinta de GEPP e precisa ser catalogada.

Tabela 2.2 – Formas matriciais dos núcleos BLAS.

m	n	k	Ilustração	Nome
grande	grande	grande		GEMM
grande	grande	1		GER
grande	grande	pequeno		GEPP
grande	1	grande		GEMV
grande	pequeno	grande		GEMP
1	grande	grande		GEVM
pequeno	grande	grande		GEPM
grande	pequeno	pequeno		GEPB
pequeno	grande	pequeno		GEBP
pequeno	pequeno	grande		GEPDOT

Fonte: Geijn e Quintana–Ortí [21].

De acordo com Goto e Geijn [22], a maneira de adaptar um núcleo GEMM a um sistema é separá-lo em várias das operações supracitadas, de tal forma que as matrizes de entrada sejam divididas em blocos. Dessa maneira, elas ficam separadas em vários conjuntos de dados menores que caibam, cada um por vez, inteiros nas memórias de melhor desempenho. Com isso, o processador pode realizar uma operação inteira sem ter que fazer consultas às

Tabela 2.3 – Nomenclatura dos possíveis núcleos envolvidos em GEMM.

Letra	Forma	Descrição
M	Matriz	As duas dimensões são grandes ou desconhecidas.
P	Painel	Uma das dimensões é pequena.
B	Bloco	As duas dimensões são pequenas.
V	Vetor	Uma das dimensões é 1.

Fonte: Geijn e Quintana–Ortí [21].

memórias de níveis mais altos (pior desempenho).

Outro detalhe elucidado pelo trabalho de Goto e Geijn [22] é a preocupação com a memória virtual, que passou a integrar a hierarquia de memórias, mas era ignorada em implementações BLAS. O artigo mostra que a diferença dos tempos de erros de acesso à *Translation Lookaside Buffer* (TLB) em relação a erros de acesso às *caches* L1 e L2, é em ordens de magnitude maior para os de TLB. Ao considerar esse fator na modelagem de uma nova solução, conseguiu desempenho real próximo do pico de desempenho teórico em variadas arquiteturas de processadores, e resultou na disponibilização da GotoBLAS.

Em meados da primeira década do século XXI, os processadores passaram a ser compostos por mais de um núcleo de processamento no mesmo *chip*, independentes entre si mas dividindo a memória principal (e um nível *cache*, o L3, na maioria dos casos). O funcionamento desse tipo de arquitetura é similar à existência de dois ou mais processadores iguais em um único sistema, também conhecida como *Symmetric Multiprocessor* (SMP). Com a popularização desse tipo de sistema, novas técnicas de programação foram desenvolvidas para explorar o potencial computacional para álgebra linear.

Neste sentido, Chan et al. [27] desenvolveram um escalonador de tarefas para algoritmos de álgebra linear que distribui tarefas entre os núcleos do processador dinamicamente. O escalonador utiliza a biblioteca FLAME, do próprio autor, e funciona automaticamente para todos os programas que são baseados nela. Essa biblioteca é uma reinvenção da LAPACK, mas utiliza BLAS, portanto seu escalonador não divide GEMM entre processadores.

Já D'Alberto e Nicolau [25] desenvolveram um algoritmo recursivo para decompor GEMM em tarefas distintas e de variados custos computacionais. Essa abordagem se mostrou adequada para multiplicações de dimensões matriciais elevadas (≥ 1500), e os resultados são ainda melhores em arquiteturas SMP. Por fim, Rünger e Schwind [26] juntaram todas as técnicas supracitadas, decomposição recursiva de GEMM, escalonador de tarefas, núcleos GExx (substitui-se 'xx' pelas letras da Tabela 2.3 para formar as combinações mostradas pela Tabela 2.2) otimizados em nível de instruções de processador, e *auto-tuning*. Mostrou resultados melhores que as implementações anteriores e competitivos com a MKL, biblioteca proprietária da Intel feita sob medida para seus próprios processadores.

2.1.3 Computadores em *Clusters*

Apesar do significativo aumento de desempenho de microprocessadores e do constante aperfeiçoamento das rotinas matriciais entre o período de 1980 a 2000, aplicações matemáticas necessitavam de maior poder computacional. Partindo desse pressuposto e paralelamente ao refinamento de GEMM, BLAS, LAPACK e outras bibliotecas para microprocessadores, foi criada a ScaLAPACK [28]. Esta biblioteca é uma implementação parcial da LAPACK para sistemas heterogêneos baseados em *clusters* de computadores, o que implica em memórias distribuídas conectadas por um sistema de troca de mensagens, como MPI [29] ou PVM [30].

Dentre as funcionalidades da ScaLAPACK estão escalonamento de tarefas e comunicação entre processadores [31]. Apesar de já existir antes da popularização de sistemas SMP e GPGPU (que será abordado na seção 2.1.4), as técnicas empregadas nessa biblioteca não se aplicaram a esses novos ambientes. Isso se deve às diferenças de desempenho dos múltiplos núcleos de processamento de cada estilo arquitetural e principalmente pelo alto custo de comunicação em *clusters*. Inclusive este é o motivo da implementação das rotinas de LAPACK ser parcial, pois nem todas tem ganho de desempenho.

2.1.4 Computação de Propósito Geral em GPU

No período entre os anos 2000–2010, o ritmo de evolução do desempenho dos microprocessadores – a partir de agora apenas CPU – diminuiu. De acordo com a informal Lei de Moore, deveriam dobrar o desempenho a cada 18 meses enquanto mantinham o preço. Essa meta foi atingida durante décadas principalmente através da miniaturização, até que esta chegasse próxima de seus limites físicos. Neste mesmo momento, outro tipo de processador, as GPUs, passaram a obter evolução no desempenho em um ritmo que dobrava a cada ano [2].

As GPUs foram criadas para acelerar tarefas da computação gráfica, especificamente a renderização, que é a representação de um modelo em três dimensões em uma imagem de duas dimensões. Esta envolve a definição da cor de cada ponto que forma a imagem, sendo semelhante entre eles mas independente. Inicialmente, disponibilizavam apenas funções fixas, mas adiante incorporaram programabilidade para permitir a criação de efeitos gráficos personalizados. Isso teve o efeito colateral de possibilitar também a sua utilização para outras aplicações, dando início à computação de propósito geral em GPU, ou simplesmente GPGPU [32].

Devido à natureza de semelhança e independência entre as milhares ou milhões de tarefas da renderização, o modelo arquitetural adotado para processadores GPGPU foi o *Single Instruction Multiple Threads* (SIMT) [33, 34, 35]. Tal modelo é definido por centenas ou milhares de unidades de operações aritméticas, vetorizadas, que executam uma instrução em múltiplos contextos concorrentemente.

Como previsto por Luebke et al. [32] e evidenciado em Ohshima et al. [5],

o modelo de programação original para GPU era orientado a aplicações gráficas. Para uma aplicação de propósito geral ser possível, o conjunto de dados do problema tinha que ser descrito em forma de textura, e o algoritmo ser descrito em forma de programa *shader*, que processa as imagens. Esse modelo dificultava a abstração e inseria complexidade desnecessária à programação. Como resposta ao problema, surgiram diversas ferramentas como em Buck et al. [36] e Mark et al. [37], até que CUDA e OpenCL se definissem como os modelos preferíveis.

Apesar de alguns resultados expressivos em determinadas aplicações, para várias outras o desempenho era similar ou inferior aos das CPUs. Este quadro começou a mudar com a incorporação de hierarquia de memórias às arquiteturas das GPUs. Esse fato, combinado com a fácil utilização e conseguinte popularização permitida pelas ferramentas de produção supracitadas, levou a vários estudos indicando ganhos em desempenho de 10 a 1000 vezes em relação às CPUs. No entanto, Lee et al. [38] mostra que essas razões são irreais, baseadas em comparações malfeitas entre códigos pouco otimizados. Ainda assim, o trabalho mostra que há ganhos reais na maioria dos casos analisados, mesmo que de somente 1,5 a 5 vezes.

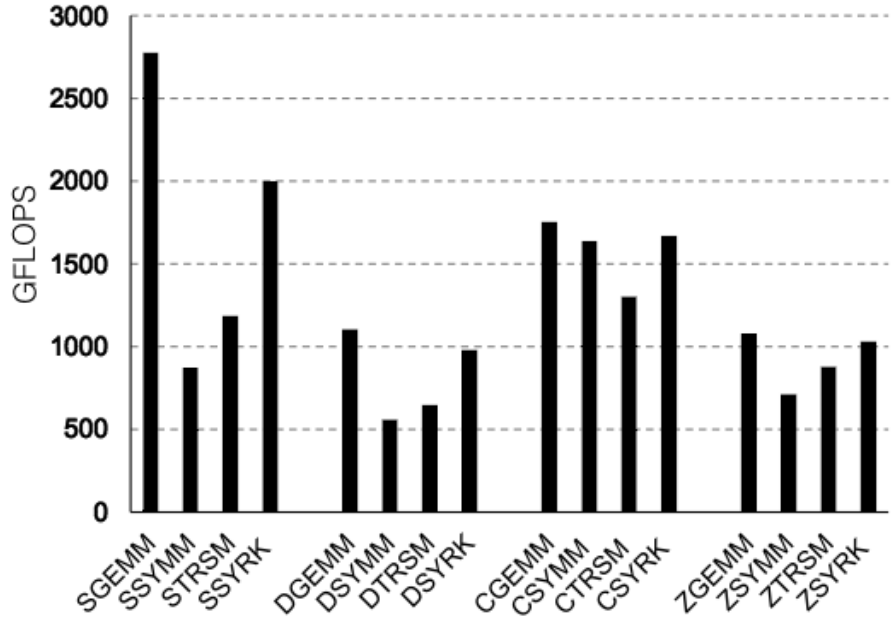
2.1.5 Multiplicação Matricial em GPGPU

Dentre as aplicações GPGPU com grande ganho de desempenho em relação às CPUs estão algumas da álgebra linear, como SAXPY e GEMM [38]. Pelo fato deste último ser um núcleo chave na obtenção de desempenho em diversos algoritmos, como discutido anteriormente, se tornou um dos principais objetos de pesquisa.

Como é uma operação altamente paralelizável e com certa independência entre as iterações, de maneira similar à renderização, a multiplicação matricial tem potencial para ser processada a taxas próximas da capacidade máxima das GPUs [39]. A dificuldade para tal está em eliminar problemas de latência inerentes à essa arquitetura e a correta utilização das memórias da hierarquia, quando possível. Diferentemente do que ocorre em CPUs, cada memória é gerenciada manualmente pelo programador.

O trabalho pioneiro de Volkov e Demmel [6] mostrou de maneira detalhada a relevância dessas questões ao elevar a taxa de uso da capacidade de uma arquitetura GPU de cerca de 45% para 66%. Tal feito foi realizado ao revolucionar a utilização dessa classe de processadores, tratando-os como se fossem uma classe especial de processadores vetoriais e de múltiplos núcleos, e principalmente levando em conta a capacidade de cada nível de memória. Outros trabalhos foram inspirados por este e utilizaram essa abordagem para desenvolver novas técnicas que permitiram evoluções no desempenho de GEMM para GPGPU [40].

Outra questão relevante anotada por Volkov e Demmel [6] e confirmada em todos os outros trabalhos é o *threshold* (ponto de partida, do inglês) do desempenho de GEMM em GPUs. Por conta da alta latência do escalonador de tarefas que controla as unidades aritméticas, a inicialização de execução de um programa é lenta. Por conta disso, multiplicações



Fonte: CUDA Toolkit 5.0 Performance Report. Disponível em: <https://developer.nvidia.com/content/cuda-5-performance-report-now-available/>. Acesso em: 20 jan. 2013

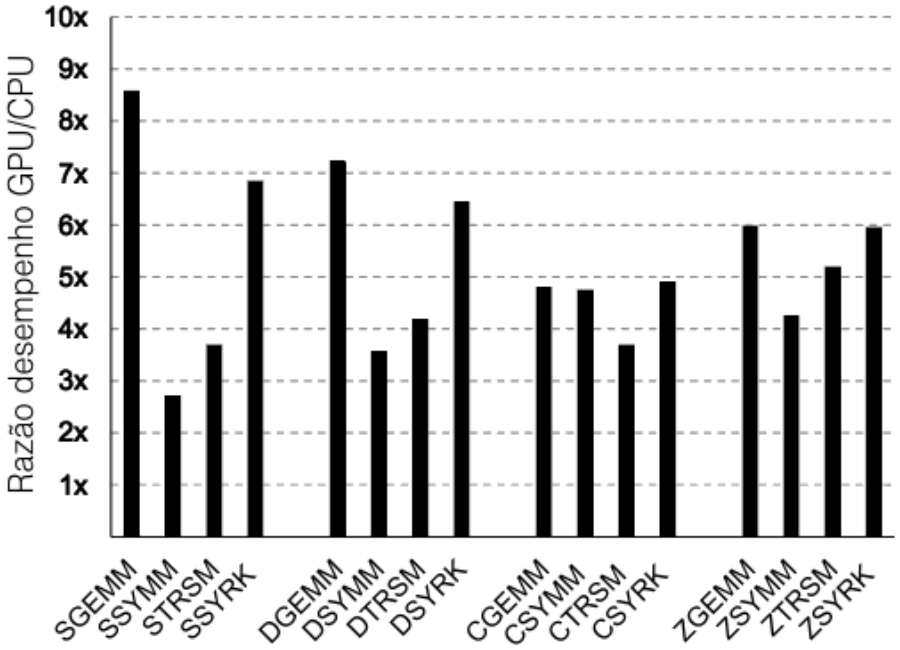
Figura 2.1 – Desempenho em GFLOPS da GPU Tesla K20X executando núcleos da CUBLAS 5.0.

matriciais menores que 500×500 atingem cerca de apenas 10–20% do desempenho dessa mesma operação sobre matrizes 4000×4000, sendo mais lentas que CPUs.

Nakasato [39] e Matsumoto et al. [41] foram além ao introduzirem o conceito de ocupação das unidades aritméticas das GPUs. Por inerência ao estilo arquitetural, se uma unidade é capaz de executar *N threads* por instrução mas só é abastecida com dados para uma única *thread*, o tempo de execução é o mesmo que se tivesse com ocupação completa. Portanto, a organização das matrizes de entrada, além de subdivididas para caberem em cada nível de memória interna, deve levar em conta a ocupação das linhas de execução. Para tal, devem ser feitas considerações sobre latência de memórias, modelo de acesso e quantidade mínima de instruções.

Com o desenvolvimento de algoritmos e técnicas específicas para execução de GEMM e, consequentemente, outros núcleos BLAS em GPU, as bibliotecas BLAS especializadas nessa tecnologia passaram a superar as de BLAS em CPU. A Figura 2.1 mostra o desempenho máximo, em GFLOPS, de alguns núcleos BLAS da biblioteca CUBLAS, da NVIDIA, para GPUs, dentre eles os da família GEMM. A Figura 2.2 mostra o ganho de desempenho desses mesmos núcleos comparados a seus equivalentes da biblioteca MKL, da Intel, que possui o melhor desempenho em CPUs. O sistema utilizado para a comparação foi composto por CPU *Intel SandyBridge* E5-2687W, GPU *Tesla Kepler* K20X e as versões mais recentes das bibliotecas.

Um fator agravante na busca pelo desempenho em GPU são as diferenças



Fonte: CUDA Toolkit 5.0 Performance Report. Disponível em: <https://developer.nvidia.com/content/cuda-5-performance-report-now-available/>. Acesso em: 20 jan. 2013

Figura 2.2 – Desempenho de alguns núcleos da CUBLAS em relação aos da MKL.

arquiteturais entre os vários modelos, ainda que de um mesmo fabricante. Apesar de seguirem o mesmo conceito estrutural, as diferenças de implementação são suficientes para fazer com que o melhor algoritmo projetado especificamente para um modelo atinja apenas uma fração de seu desempenho em outros. Outro agravante é o fato de que as melhores implementações de GEMM são programadas em linguagem intermediária [39, 34], direcionadas a algum modelo específico.

Em Du et al. [35], verifica-se a incapacidade de portabilidade do desempenho, apesar da alta portabilidade de um algoritmo implementado em CUDA ou OpenCL. Ou seja, uma implementação GEMM pode ser recompilada para praticamente todos os modelos existentes sem alterações no código, mas o desempenho não acompanha. Dessa forma, buscou-se a solução já utilizada com sucesso em núcleos matriciais para CPU, o *auto-tuning*. Enquanto o próprio trabalho de Du et al. [35] apenas apontou alguns parâmetros a serem considerados, Cui et al. [42] e, principalmente, Kurzak et al. [43] fizeram implementações com relativo sucesso. O primeiro utilizou como parâmetro apenas a ocupação de memórias internas, enquanto o segundo considerou também a ocupação de *threads*, apesar de alguns resultados contraditórios, já que não demonstraram melhoras unânimes, apenas pontuais.

Outra questão técnica com influência no projeto de algoritmos é a quantidade de memória disponível, pois, da forma como são construídos e utilizados, os modelos de GPUs vêm de fábrica com uma quantidade pré-definida que não pode ser expandida. Além disso, essa memória não está integrada por *hardware* à hierarquia de memórias principal do sistema, e não tem acesso a memória virtual. Isso significa que o espaço é limitado e o sistema não abstrai

essa condição para o programador.

Então, um núcleo GEMM para GPGPU, em condições naturais, possui limites para o tamanho das matrizes de entrada. Para resolver essa situação, Sun e Tong [44] desenvolveram algoritmos para decomposição de uma GEMM enorme, segundo o próprio trabalho, envolvendo matrizes maiores que 8000×8000, em várias GEPP (ver Tabela 2.2). Dessa forma, cada uma dessas operações menores é executada independentemente e retornam parte do resultado final, montado aos poucos.

A decomposição de GEMM é importante também para sistemas compostos por mais de uma GPU. Pelo fato de não haver um hardware escalonador dedicado a distribuição de tarefas entre si, nos mesmo moldes de uma arquitetura SMP, cada núcleo deve ser executado explicitamente em um dos processadores GPGPU disponíveis. Sun e Tong [44] e Volkov e Demmel [6] tratam do assunto para GEMM e fatoração *Lower-Upper* (LU) [10], respectivamente, verificando quantas GPUs estão disponíveis no sistema e dividindo o trabalho igualmente entre elas, através de escalonamento estático.

No entanto, desde a decomposição de algoritmos de álgebra matricial para sistemas SMP, sabe-se que a abordagem ideal para escalonamento de tarefas entre múltiplos processadores homogêneos é a realizada dinamicamente. Isso se deve ao fato desta forma de escalonamento ser capaz de manter funcionamento e desempenho mais uniformes quando utilizada por sistemas diversos e em aplicações com variações de tarefas, como é o caso em sistemas de produção [27]. Quintana-Ortí et al. [45] adapta o escalonador dinâmico para SMP, de trabalhos anteriores, para funcionar com GPGPU e mostra ser um caminho para o desempenho, além de maior praticidade de programação. O porém é que a divisão de tarefas não se dá em nível de BLAS, ou seja, não separa GEMM em núcleos menores, ou seja, multiplicações matriciais não são subdivididas para resolução em múltiplos processadores simultaneamente.

Uma consideração a ser feita quando se utiliza processadores aceleradores, que possuem espaço de memória própria, e são ligados ao processador central por barramentos de dados, é o desempenho das transferências de dados entre eles. Este é o caso de sistemas com GPGPU, que são compostos por uma ou mais GPUs e uma CPU, sendo esta o centro das iniciativas. Isso significa que a Entrada e Saída (E/S) de dados sempre ocorre através da unidade central. Caso uma tarefa seja delegada a um acelerador, os dados devem ser transferidos para ele, para então serem processados e, na maioria dos casos, transferidos de volta.

O fato é que tais operações de transferências de dados entre os processadores com memória distribuída são lentas quando comparadas à capacidade de obter altas taxas de processamento de dados. Os processadores são capazes de tratar mais dados do que os barramentos são capazes de prover, para casos em que cada unidade do conjunto de dados seja acessada poucas vezes. Alvarez et al. [46] mostram que, ao eliminar transferências de dados pelo barramento em momentos chave de algoritmos recursivos, que envolvem várias dessas operações, o desempenho aumentou em até 6 vezes.

Ao considerar essa questão para o caso específico de GEMM, Allada et al. [47] verificaram que as operações de transferências entre memórias da CPU e GPU ocupam um tempo significativo. Elas são relevantes na definição do desempenho global de GEMM em GPGPU, ao considerar o processo inteiro do ponto de vista da CPU. O trabalho também utiliza uma ferramenta de medição de desempenho de operações de memória, a NetPIPE [48], para verificar que as ferramentas de medição providas por CUDA são confiáveis. A necessidade de otimizar as transferências pelo barramento de dados também é abordada em Fatica [49], que adaptou rotinas da LINPACK e GEMM para GPGPU em computadores de produção.

2.1.6 Sistemas Heterogêneos

Quando um sistema computacional é composto por processadores de configurações distintas, mesmo que sejam mudanças pontuais como diferença na frequência do ciclo ou quantidade de *cache* L2, ele é considerado um sistema ou ambiente heterogêneo. No entanto, apesar dessa definição abrangente, o termo tem sido utilizado para se referir a sistemas compostos por CPUs e aceleradores como GPUs [5, 50]. Outro termo utilizado para este mesmo caso é "sistema híbrido" [7, 35].

Os primeiros trabalhos a explorarem a heterogenia em GEMM e álgebra matricial apenas buscavam a utilização de todos os núcleos de processamento disponíveis em um sistema, para que a soma entre eles resultasse em desempenho superior. Eles dividiam as operações independentes entre os processadores, para simular um funcionamento similar ao de SMP [5, 49, 45, 44].

Sistemas SMP têm como características principais a existência de ao menos dois tipos de processadores com afinidade a tarefas distintas, a memória distribuída e o barramento de dados interligando-os. Tipicamente, a CPU possui núcleos de processamento de baixa latência, ideais para resolver um conjunto de instruções o mais rapidamente possível. Já as GPUs possuem um alto grau de paralelismo, e são ideais para operar sobre múltiplos dados de maneira uniforme. Dessa forma, as CPUs apresentam desempenho superior em instruções de lógica, desvios e operações pontuais, enquanto as GPUs superam em atualização de grande quantidade de dados e operações dependentes de grande movimentação de memória [38].

Por essa diferença, muitos trabalho levaram em conta o ambiente heterogêneo ao distribuir tarefas adequadamente a cada processador. Volkov e Demmel [6], Tomov et al. [7] desenvolveram algoritmos com distribuíção de cargas entre os distintos processadores disponíveis de acordo com a afinidade. O primeiro programou um método para fatoração LU em que os passos lógicos são deixados para a CPU e as multiplicações matriciais são divididas entre duas GPUs. O segundo iniciou a produção da biblioteca MAGMA [51], uma implementação das interfaces de LAPACK que visa dividir tarefas das rotinas entre uma única GPU e a CPU.

Porém, uma vantagem de sistemas heterogêneos baseados em GPGPU é a escalabilidade da capacidade computacional ao adicionar mais GPUs ao sistema. E ao utilizar qualquer das abordagens supracitadas para implementação das rotinas, os algoritmos são

incapazes de se beneficiar dessa característica. Por isso, é importante utilizar uma abordagem de listas de tarefas, onde um problema é decomposto em tarefas que são armazenadas em uma estrutura de dados. A distribuição das tarefas entre os processadores é feita por um escalonador dinâmico que conheça ambos os lados, ou seja, a natureza da tarefa e a afinidade de cada processador. Esta é a abordagem utilizada nos trabalhos de Igual et al. [1] e Bosilca et al. [52].

2.2 TRABALHOS CORRELATOS

A adequação de algoritmos matriciais para sistemas habilitados à GPGPU acontece de maneira acelerada, e o mesmo pode-se dizer para sistemas heterogêneos. No entanto, poucos trabalhos se dedicam a adequar o núcleo mais importante a esses algoritmos, o GEMM. A maioria implementa multiplicação matricial para uma única GPU e a utiliza em várias GPUs apenas quando uma rotina LAPACK prevê varias multiplicações que podem ser paralelas [49, 7, 1].

Dentre os trabalhos que buscam a heterogenia no processamento de GEMM, está Ohshima et al. [5]. Ele formaliza a carga computacional de GEMM e a capacidade teórica de processamento de um sistema composto por uma GPU e uma CPU. Dessa forma, busca a exata divisão de uma GEMM em duas *General Panel-Matrix Multiplication* (GEPM) de tal forma que ambos processadores levem o mesmo tempo na tarefa. Apesar do relativo sucesso, sabe-se que não é trivial prever o desempenho de um sistema, o que explica as imperfeições no resultado apresentado. Além disso, não leva em conta a retenção de desempenho causada por transferências de memória. Um último problema é que as GPUs estão evoluindo de tal forma que a necessidade de ter a CPU dividindo a multiplicação é questionável, em vista de que em 2012, a melhor CPU atinge cerca de 200 GFLOPS, enquanto a melhor GPU atinge até 2,5 TFLOPS.

Em Fatica [49], outro trabalho sobre o tema, GEMM é divido n GEPMs, em que n é a quantidade de GPUs disponíveis no sistema. Apesar de representar uma abordagem mais genérica, ainda é pouco, leva em conta poucas das características relevantes de um sistema heterogêneo. O trabalho não considera a escalabilidade do custo de transferências entre memórias pelo barramento em relação ao aumento da quantidade de GPUs participantes da operação, ignora o fato de que para cada GPU adicionada, deve-se transferir para ela dados o suficiente para o processamento. Também não há no trabalho uma análise da melhor maneira de se dividir GEMM de tal forma a minimizar tais transferências, como propõem Tomov et al. [7].

Em Silva et al. [53], trabalho deste autor, foi desenvolvida uma nova abordagem para a decomposição de GEMM e distribuição das tarefas entre processadores de um sistema formado por uma CPU e duas GPUs. A divisão é feita utilizando o subnúcleo GEPDOT e as tarefas são classificadas de forma a evitar redundância nas transferências de dados entre os processadores, com o objetivo de reduzir o tempo gasto nesta operação, que foi identificada como um dos pontos de retenção de desempenho. Dessa forma, o trabalho

mostrou que novas abordagens são possíveis para aumentar o desempenho geral, e evidenciou a necessidade de analise dos distintos e possíveis algoritmos de decomposição de GEMM para ambientes heterogêneos.

3 PROGRAMAÇÃO PARA SISTEMAS HETEROGÊNEOS

Programar para ambientes heterogêneos compostos por múltiplos núcleos de CPUs organizados em SMP, e adicionados por uma ou mais GPUs, requer o conhecimento de novos conceitos. Enquanto o modelo SMP garante um contexto computacional para programação que é similar ao de CPUs com apenas um núcleo, a adição de coprocessadores GPGPU requer um contexto diferenciado.

Ao projetar algoritmos para o contexto heterogêneo, é necessário distinguir as unidades computacionais entre hospedeira e dispositivos. A hospedeira é a unidade central de processamento, que manipula a entrada e saída de dados, é responsável pela memória principal e é o ponto de vista dos algoritmos principais, que gerenciam a execução de sub-rotinas (ou sub-algoritmos) nos dispositivos. Os dispositivos são as unidades de coprocessamento, que possuem arquitetura distinta em relação à hospedeira, e, por esse motivo, não pode ser integrada no modelo SMP, além de exigir programação de rotinas próprias. Por serem unidades capazes de manipular quantidades de dados maiores que as CPUs, requerem memória própria e de maior desempenho.

Dessa forma, o contexto de programação heterogêneo tem as CPUs em SMP no papel de hospedeira e as GPUs como dispositivos. Os algoritmos principais têm como ponto de vista a hospedeira e gerenciam explicitamente a execução de algoritmos secundários nos dispositivos. Como cada uma dessas unidades computacionais desse contexto possui seu próprio espaço de memória, o algoritmo principal também gerencia explicitamente o fluxo de dados entre eles. Além disso, como o funcionamento dos elementos é concorrente entre eles, cabe ao algoritmo principal a definição e sincronização das *threads*.

Este capítulo define e explica a utilização das ferramentas existentes para as tarefas de gerenciamento do contexto heterogêneo. A organização das seções é a seguinte: a seção 3.1 discute o gerenciamento de memória em ambientes heterogêneos; a seção 3.2 explica o conceito de lotes de execução, utilizado em programação GPGPU, e a forma de expressão das operações empregada neste trabalho.

3.1 GERENCIAMENTO DAS MEMÓRIAS

A principal característica de ambientes heterogêneos é possuir processadores distintos em um único sistema. Este trabalho se preocupa principalmente com os casos em que o sistema é formado por uma CPU tradicional e por uma ou mais GPUs, ligadas a CPU por meio de um barramento de dados de alto desempenho. Neste caso, cada processador possui sua própria memória principal, e se configura como um sistema de memória distribuída.

Como cada unidade funciona independentemente, há um paralelismo em nível

de processadores. Para que trabalhem sobre um mesmo conjunto de dados e, portanto, em um mesmo problema, tais dados devem ser modelados de forma que possam ser divididos. Em seguida, são distribuídos entre cada processador participante na resolução através de transferência de dados pelo barramento. Após a resolução, os dados devem ser transferidos de volta à CPU.

O problema é que transferências de dados entre CPU e GPUs são limitadas pela banda do barramento de dados que faz a ligação entre eles. Esta banda, apesar de ser de alto desempenho, quando comparada a outras formas de conexão, possui apenas uma fração do desempenho das memórias das GPUs e CPUs. Por esse motivo, são sujeitas a serem os principais pontos de retenção do desempenho na execução de algoritmos GPGPU.

Por exemplo, em 2012, uma única GPU era capaz de processar alguns TFLOPS em condições teóricas ideais, e a comunicação com sua memória principal atingia taxas de $\sim$ 300GB/s (*gigabytes* por segundo). Já a conexão entre CPU e GPU no melhor caso, utilizando barramento *Peripheral Component Interconnect Express* (PCIe) 3.0 e uma quantidade suficiente de placas GPU para saturá-lo, possui desempenho ideal estimado de apenas 16GB/s. Por este motivo as transferências de dados entre memórias de processadores de um sistema heterogêneo devem ser minimizadas, com a intenção de melhor aproveitar o potencial computacional.

3.1.1 Ciclo de Vida de Objetos de Memória

Em sistemas comuns, objetos de memória têm seus espaços alocados na memória principal e permanecem lá enquanto forem necessários, até serem liberados. Já em ambientes heterogêneos, esse ciclo de vida da memória ocorre de maneira distinta. Objetos de memória têm seu espaço alocado primeiramente no hospedeiro para então, quando for necessário na execução de um *kernel* GPGPU, ter mais um espaço alocado no dispositivo a realizar a operação. Em seguida, seus dados são transferidos de uma memória à outra para que o espaço no dispositivo seja réplica do espaço no hospedeiro. Após o processamento, os dados são transferidos de volta, caso seja necessária a atualização da cópia do hospedeiro, e o espaço de memória no dispositivo é liberado. Por fim, quando a utilização no sistema acabar, o espaço de memória no hospedeiro é liberado. A Figura 3.1 ilustra esse ciclo de vida dos objetos de memória em ambientes heterogêneos.

Para simplificar, os algoritmos expostos neste trabalho omitem alocações e liberações de memória, partem do pressuposto que elas já tenham sido feitas corretamente, apesar de o gerenciamento de memória ser necessário e explícito quando se utiliza as *Application programming interface*s (APIs) CUDA e OpenCL. Já as transferências entre memórias, que estão entre os alvos de estudo deste trabalho, são indicadas explicitamente com as notações $X \rightarrow D_i$ e $X \rightarrow H$, que significam, respectivamente, que um objeto de memória X foi transferido do hospedeiro para o dispositivo D_i, e de algum dispositivo para o hospedeiro H.

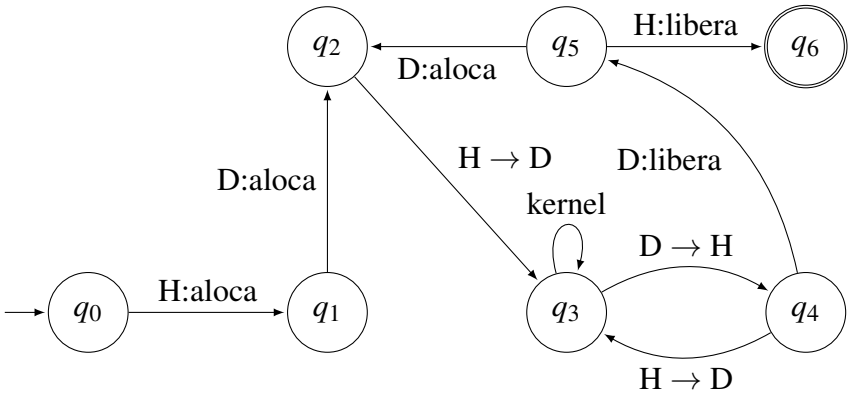


Figura 3.1 – Ciclo de vida de um objeto de memória em ambientes heterogêneos.

3.1.2 Estudo de Caso: GEMM GPGPU

Para fim de ilustração do problema, toma-se o sistema descrito pela tabela 3.1 para execução da rotina *Single precision GEMM* (SGEMM) provida pela biblioteca clAmdBlas em uma única GPU. A tabela exibe as seguintes especificações técnicas: CPU; *Read-Only Memory* (RAM); Sistema Operacional (SO); *Graphics Bus*, a tecnologia utilizada para barramento de dados entre CPU e GPU; GPU; e, por fim, *Video RAM* (VRAM).

Neste caso, multiplica-se duas matrizes de Precisão Simples (PS) (4 bytes por elemento) com tamanho de 1000×1000, o que resulta em um conjunto de dados de entrada de 8 *Megabyte*s (MBs), ou $1000 \times 1000 \times 4 \times 2$. Sabe-se através de testes anteriores que neste caso o processador utilizado atinge uma taxa de processamento de ~ 750 GFLOPS PS e que o sistema possui taxas teóricas de transferências de dados de 5,15 *Gigabytes* por segundo (GB/s) de CPU para GPU, e 5,87 de GPU para CPU. O conjunto de dados matriciais inicia-se no espaço de memória da CPU, é transferido para o espaço da GPU, é processado e então é transferido de volta na forma de uma única matriz 1000×1000, com tamanho total de 4 MBs.

A medição dos tempos de execução é exibida na figura 3.2. Nesse caso específico, as transferências de memória ocupam pouco mais da metade do tempo total de execução da rotina. Por tanto, uma das maneiras de ganhar desempenho é otimizar esse tipo de operação, seja na implementação de algoritmos que minimizem a comunicação entre os processadores, seja nos detalhes técnicos de utilização do sistema.

3.1.3 Técnicas de Otimização

A melhor maneira de diminuir a influência das transferências de memória no desempenho final é evitá-las ao máximo. No entanto, como é inevitável que em algum momento sejam necessárias, a idéia é que ocorram da melhor maneira possível, gerando o menor impacto possível. Algumas condições do sistema e algumas características dos dispositivos são capazes de influenciar positivamente nas transferências, diminuindo seu tempo operacional. Dentre elas estão o *Direct Memory Access* (DMA), o *cache* da CPU e a sobreposição de transferências.

Tabela 3.1 – Configuração do sistema utilizado na medição dos tempos de SGEMM da clAmdBlas.

Tipo	Nome	Notas
CPU	Core i5 760 @ 2.8GHz	80 GFLOPS PD
RAM	4GB DDR3 1333MHz	21 GB/s
SO	Windows 7 Pro	x86_64
Graphics Bus	PCIe 2.0	8 GB/s
GPU	Barts Pro @ 820MHz	1488 GFLOPS PS
VRAM	1GB GDDR5 1050MHz	128 GB/s

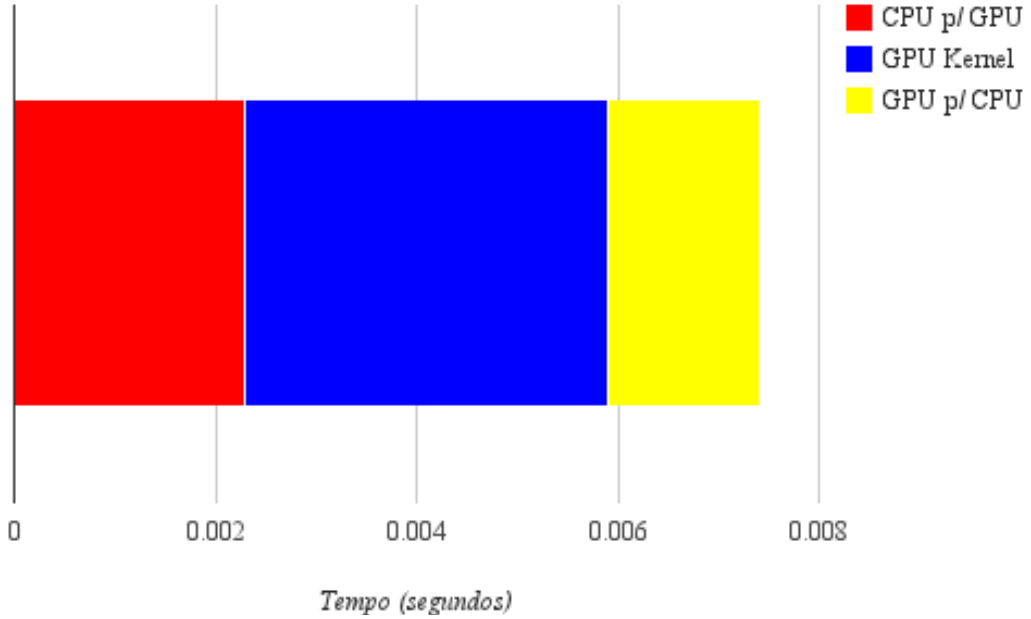


Figura 3.2 – Medição da rotina SGEMM da clAmdBlas para matrizes 1000×1000.

3.1.3.1 O DMA

Para que um objeto de memória seja transferido entre CPU e GPU, o espaço pertencente à CPU deve estar habilitado ao DMA. Para que este seja o caso, o objeto de memória deve estar corretamente alinhado em relação à GPU e deve estar em página fixa da memória.

O alinhamento se refere ao posicionamento espacial do objeto na memória de tal forma que seus elementos possam ser carregados pelos registradores de um processador sem a necessidade de realizar deslocamentos de bits e aplicação de máscaras que remontem o dado inteiro em apenas um registrador. O correto alinhamento de um objeto deve ocorrer no momento de sua criação, utilizando as chamadas adequadas e específicas a cada sistema.

A página fixa de memória é aquela que nunca sai do campo de visão principal da CPU, ou seja, nunca é colocada na memória virtual e não é trocada em mudanças de processos do sistema. Um processo só está disponível para DMA caso esteja nessa página fixa. No entanto, é um recurso escasso, que pode ter o acesso negado, e sua utilização indiscriminada

pode acarretar em baixo desempenho de outros processos concorrentes existentes no sistema.

A falta de alinhamento e o não posicionamento de objetos de memória na página fixa geram uma operação de movimentação de dados extra por parte da CPU para garantir essas condições no momento de cada transferência de dados entre hospedeiro e dispositivo. Portanto, sempre que possível, objetos de memória já devem ser criados com essas condições garantidas. No entanto, enquanto garantir o alinhamento é puramente uma questão de adaptação de um dado código às chamadas de sistema adequadas, a utilização de página fixada deve ser restringida. Dessa, forma, neste trabalho considera-se que todos os objetos de memória utilizados em algoritmos já se encontram alinhados, mas a colocação em página fixada é uma operação explícita.

3.1.3.2 O *cache* da CPU

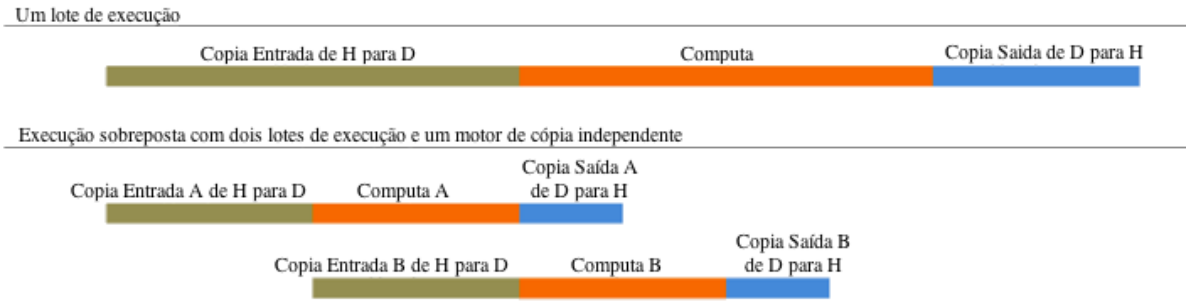
Leituras e escritas à memória principal pela CPU normalmente passam pelos seus níveis de *cache*. Porém, é possível desabilitar esse modelo de fluxo de dados, fazer com que a comunicação seja direta entre registradores e memória principal. Nesse modelo, a leitura de dados é mais lenta por ordens de magnitude, mas a escrita é mais rápida em até 40% [54, 55]. Utilizar essa forma de escrita combinado com o acesso direto à memória da GPU por parte da CPU pode ser útil para as transferências de objetos no sentido desta para aquela. No entanto, essa tarefa mantém a CPU ocupada e a criação do objeto de memória deve utilizar uma chamada de sistema específica. Por isso e por ter utilidade apenas pontual, tal recurso não foi utilizado neste trabalho.

3.1.3.3 A sobreposição de transferências e *kernels*

Algumas GPUs, especialmente as destinadas à produção profissional e científica, são capazes de sobrepor operações de transferências de objetos de memória e execução de *kernels* GPGPU. Em outras palavras, essas duas operações acontecem simultaneamente, o que é uma característica significativa na busca por melhor desempenho e, portanto, deve ser considerada em projetos de algoritmos. Normalmente, um conjunto de dados é transferido por inteiro em uma única operação de transferência e em seguida processado por um *kernel* GPGPU. Para explorar as transferências sobrepostas, este conjunto de dados deve ser divido em ao menos dois menores, para que o primeiro seja transferido, e comece a ser processado ao mesmo tempo em que se inicia a transferência do segundo, além de ser necessária a utilização de ao menos dois lotes de execução, abordados na seção 3.2.

A Figura 3.3 ilustra os modos de operação comuns e com o emprego de sobreposição. No primeiro caso, ocorre a transferência de dados do hospedeiros para o dispositivo, para somente então iniciar o *kernel* GPGPU e, por fim, realizar a transferência de dados no sentido oposto. Já no segundo caso, dados e *kernels* foram separados em dois grupos, A e B. No término da cópia de A entre hospedeiro para dispositivo, é iniciado o processamento desses dados ao mesmo tempo em que começa a transferência de B para o dispositivo. No

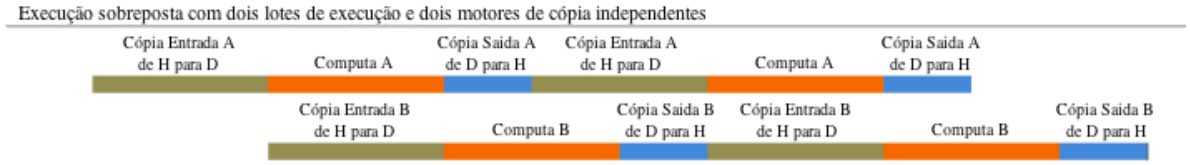
momento do término de B, é iniciada a sua computação, enquanto A, então com processamento também terminado, inicia transferência no sentido oposto. O resultado final é um tempo total de operação menor.



Fonte: [56].

Figura 3.3 – Comparação entre execuções sequencial simples e com sobreposição entre transferências e *kernels*.

Alguns dispositivos são capazes de sobrepor duas transferências de dados, uma em cada sentido, além da execução de um *kernel*. A Figura 3.4 ilustra a característica com outro caso de uso, em que ocorre uma sobreposição de transferências de entrada e saída, no momento da segunda cópia de A de hospedeiro para dispositivo, ao mesmo tempo em que termina o processamento de B, que é transferido do dispositivo para hospedeiro.



Fonte: [56].

Figura 3.4 – Sobreposição entre transferências em ambos sentidos e *kernels*.

3.2 Lotes de Execução

As GPUs são processadores caracterizados pelo paralelismo. Enquanto um núcleo de CPU é capaz de executar entre 1 à 4 *threads* simultaneamente, um de GPU executa entre 32 à 192 *threads*, a depender do modelo. Além disso, as GPU podem apresentar até 15 núcleos em um único *chip*, sendo capazes de executar milhares de *threads* simultaneamente.

Porém, problemas a serem executados em GPU precisam ter quantidade de dados suficiente para ocupar todo esse paralelismo, o que nem sempre se verifica. A solução é colocar mais de um *kernel* GPGPU para execução simultânea. No entanto, essa concorrência pode ter os mesmos problemas de *threads* em CPU que trabalham sobre um mesmo conjunto de dados, o acesso de leitura e escrita não sincronizado pode gerar corrupção. Para piorar,

a utilização de sinalização estilo *mutex* é inviável, pela baixa qualidade em desempenho nas operações de desvios condicionais por parte das GPUs.

Para resolver essa situação, a programação de GPUs utiliza lotes de execução, que são listas de tarefas organizadas em filas. Dessa forma, duas tarefas t_1 e t_2 dispostas dessa forma em um lote de execução serão executadas nessa ordem garantidamente, e não haverá concorrência entre elas. Além disso, cada GPU pode ter associada a si quantos lotes forem necessários, e as tarefas entre os diferentes lotes podem ser executadas concorrentemente, além de não haver qualquer garantia na ordem exata de resolução. Portanto, neste caso, qualquer sincronização precisa ser feita explicitamente.

Os lotes de execução aceitam três tipos de tarefas, que são:

- Transferência de dados entre hospedeiro e dispositivos;
- Execução de *kernel* GPGPU;
- Sincronização do lote;

A sincronização do lote significa que a execução da rotina no hospedeiro ficará interrompida até que o ponto de sincronização desejado no lote seja atingido. Se não houver nenhum ponto alvo, a espera perdura até que todas as tarefas do lote terminem.

Neste trabalho utiliza-se a notação

$$s \leftarrow (s,t)$$

em algoritmos para indicar que a tarefa t foi inserida no fim da fila do lote s. As operações de sincronização são indicadas explicitamente de maneira textual.

4 DECOMPOSIÇÃO DE MULTIPLICAÇÃO MATRICIAL EM TAREFAS

Este capítulo apresenta dois algoritmos para decomposição de multiplicação matricial GEMM em tarefas, e faz uma comparação entre eles. A organização é realizada da seguinte forma: a seção 4.1 desenvolve e analisa o algoritmo Dec_GEPM; a seção 4.2 desenvolve e analisa o algoritmo Dec_GEPDOT; por fim, a seção 4.3 faz uma comparação entre os algoritmos desenvolvidos.

4.1 ALGORITMO DEC_GEPM

O algoritmo Dec_GEPM é baseado no trabalho de Fatica [49], que separa um núcleo GEMM em dois GEPM, para que um seja executado na GPU e outro na CPU. Porém a abordagem do autor é a simples execução direta, não utiliza o conceito de divisão em tarefas e prevê escalonamento estático, o que torna o algoritmo específico a um tipo de sistema e inadequado a sistemas de produção, com cargas de trabalho e taxas de ocupação imprevisíveis.

Dada a multiplicação matricial GEMM $C \leftarrow \alpha AB + \beta C$, onde α e $\beta \in \mathbb{R}$, $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$ e $C \in \mathbb{R}^{m \times n}$, com m, n e $k \in \mathbb{N}^*$, divide-se A e C tal que

$$A \rightarrow \begin{bmatrix} A_1 \\ \vdots \\ A_p \end{bmatrix} \text{ e } C \rightarrow \begin{bmatrix} C_1 \\ \vdots \\ C_p \end{bmatrix},$$

em que $A_i \in \mathbb{R}^{\frac{m}{p} \times k}$ e $C_i \in \mathbb{R}^{\frac{m}{p} \times n}$, onde $p \in \mathbb{N}^*$ define a quantidade de GEPMs a serem gerados. A multiplicação matricial completa passa a ser expressa por

$$\begin{bmatrix} A_1 \\ \vdots \\ A_p \end{bmatrix} \times B = \begin{bmatrix} C_1 \\ \vdots \\ C_p \end{bmatrix},$$

que é a união de várias operações

$$C_i \leftarrow \alpha A_i B + \beta C_i$$

O grupo T de tarefas extraídas é definido por $T = \{C_1, \cdots, C_p\}$ em que cada operação envolve a mesma quantidade de dados a serem manipulados. O algoritmo 1 sintetiza este método de extração das tarefas.

4.1.1 Análise de Uso de Memória

Para a execução de uma tarefa t_i em T por um dispositivo, é necessário transferir uma quantidade de palavras de memória

$$T_{in} = A_i + B + C_i = \frac{m}{p} \cdot k + k \cdot n + \frac{m}{p} \cdot n = \frac{m}{p}(k+n) + k \cdot n$$

Algoritmo 1 Dec_GEPM

Entrada: $A \in \mathbb{R}^{m\times k}$, $B \in \mathbb{R}^{k\times n}$, $C \in \mathbb{R}^{m\times n}$, $\alpha, \beta \in \mathbb{R}$ e $p \in \mathbb{N}^*$
Saída: Tarefas $T = (t_1, \cdots, t_p)$

1: Partir $A \rightarrow \begin{bmatrix} A_1 \\ \vdots \\ A_p \end{bmatrix}$, onde $A_i \in \mathbb{R}^{\frac{m}{p}\times k}$
2: Partir $C \rightarrow \begin{bmatrix} C_1 \\ \vdots \\ C_p \end{bmatrix}$, onde $C_i \in \mathbb{R}^{\frac{m}{p}\times n}$
3: $i \leftarrow p$
4: **enquanto** $i > 0$ **faça**
5: $\quad T \leftarrow (T, C_i \leftarrow \alpha A_i B + \beta C_i)$
6: $\quad i \leftarrow i - 1$
7: **fim enquanto**
8: **retornar** T

do hospedeiro ao dispositivo, caso β do núcleo seja diferente de zero. Para $\beta = 0$, tem-se

$$T_{in} = \frac{m}{p} \cdot k + k \cdot n.$$

Já transferências de algum dispositivo para o hospedeiro é sempre

$$T_{out} = C_i = \frac{m}{p} \cdot n.$$

Um detalhe importante sobre o uso de memória desse algoritmo é a exigência de uma quantidade mínima de memória alocada nos dispositivos participantes. Para execução de uma tarefa t_i em T, é necessário alocar a quantidade de palavras $MA = A_i + B + C_i = \frac{m}{p}(k + n) + k \cdot n$. A partir dessa equação, é possível perceber que, se um dispositivo possuir quantidade de memória inferior a $B = k \cdot n$ palavras de um dado problema, este dispositivo não poderá ser utilizado em sua resolução, pois $\lim_{p\to\infty} MA = k \cdot n$. Em outras palavras, não adianta particionar o problema em mais GEPMs, pois ele nunca caberá nesse dispositivo, independente do valor de p.

Porém, uma vez que B esteja na memória de um dispositivo, a sobrecarga gerada pelas transferências do hospedeiro para o dispositivo na execução de outras tarefas em T, que necessariamente utilizam este mesmo B, é reduzida à

$$T_{in} = \begin{cases} A_i + C_i & \text{se } B \neq 0 \\ A_i & \text{se } B = 0 \end{cases}.$$

4.2 ALGORITMO DEC_GEPDOT

Nesta forma de decompor a multiplicação matricial em tarefas, parte-se da abordagem de divisão em blocos e verificação das operações entre eles, similar a Golub e Loan

[10]. Dada a multiplicação matricial $C = AB$, onde $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$ e $C \in \mathbb{R}^{m \times n}$, com m, n e $k \in \mathbb{N}^*$, as matrizes são divididas de tal forma que

$$\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix},$$

em que $A_{i,j} \in \mathbb{R}^{\frac{m}{2} \times \frac{k}{2}}$, $B_{i,j} \in \mathbb{R}^{\frac{k}{2} \times \frac{n}{2}}$ e $C_{i,j} \in \mathbb{R}^{\frac{m}{2} \times \frac{n}{2}}$. Para os casos em que alguma das dimensões de A, B e/ou C não são múltiplos de 2, realiza-se um *peeling* matricial, que é o aumento das dimensões matriciais ao adicionar vetores identidade, garantindo a dimensão desejada mas sem mudar os dados. Após a divisão em blocos, realiza-se a multiplicação matricial com essas matrizes de blocos para chegar ao seguinte conjunto de equações

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1}, \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2}, \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1}, \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2}. \end{aligned}$$

Para facilitar a visualização, representa-se os elementos $C_{i,j}$ da matriz C em termos de multiplicações matriciais menores entre $A_{i,j}$ e $B_{i,j}$, tal que

$$\begin{aligned} C_{1,1} &= M_1 + M_2, \\ C_{1,2} &= M_3 + M_4, \\ C_{2,1} &= M_5 + M_6, \\ C_{2,2} &= M_7 + M_8. \end{aligned}$$

Assim, define-se o conjunto $M = \{M_1, \ldots, M_8\}$ das multiplicações menores e $S = \{C_{1,1}, \ldots, C_{2,2}\}$ das somas a serem resolvidas para que se chegue ao resultado da operação original. As multiplicações são dadas por

$$\begin{aligned} M_1 &= A_{1,1}B_{1,1}, M_2 = A_{1,2}B_{2,1}, \\ M_3 &= A_{1,1}B_{1,2}, M_4 = A_{1,2}B_{2,2}, \\ M_5 &= A_{2,1}B_{1,1}, M_6 = A_{2,2}B_{2,1}, \\ M_7 &= A_{2,1}B_{1,2}, M_8 = A_{2,2}B_{2,2}. \end{aligned}$$

Dessa forma, classifica-se as operações envolvidas na resolução de $C = AB$ pelo método de blocos em duas classes: M multiplicações e S adições. Os elementos de ambos os grupos se tornam tarefas a serem distribuídas pelos processadores disponíveis em um sistema heterogêneo. O algoritmo 2 sintetiza este método de extração das tarefas.

4.2.1 Dependência entre Tarefas

Como discutido no capítulo 3, é desejável reduzir as transferências de memória para melhorar o desempenho computacional global do algoritmo. Uma forma de

Algoritmo 2 Dec_GEPDOT

Entrada: $A \in \mathbb{R}^{m\times k}$, $B \in \mathbb{R}^{k\times n}$, $C \in \mathbb{R}^{m\times n}$, $\alpha, \beta \in \mathbb{R}$ e $p \in \mathbb{N}^*$
Saída: Multiplicações $M = (M_{1,1,1}, \cdots, M_{p,p,p})$ e somas $S = (S_{1,1,1}, \cdots, S_{p,p,p})$

Partir $A \rightarrow \begin{bmatrix} A_{1,1} & \cdots & A_{1,p} \\ \vdots & \ddots & \vdots \\ A_{p,1} & \cdots & A_{p,p} \end{bmatrix}$, onde $A_{i,j} \in \mathbb{R}^{\frac{m}{p}\times\frac{k}{p}}$

Partir $B \rightarrow \begin{bmatrix} B_{1,1} & \cdots & B_{1,p} \\ \vdots & \ddots & \vdots \\ B_{p,1} & \cdots & B_{p,p} \end{bmatrix}$, onde $B_{i,j} \in \mathbb{R}^{\frac{k}{p}\times\frac{n}{p}}$

Partir $C \rightarrow \begin{bmatrix} C_{1,1} & \cdots & C_{1,p} \\ \vdots & \ddots & \vdots \\ C_{p,1} & \cdots & C_{p,p} \end{bmatrix}$, onde $C_{i,j} \in \mathbb{R}^{\frac{m}{p}\times\frac{n}{p}}$

para $i \leftarrow 1$ **até** p **faça**
    para $j \leftarrow 1$ **até** p **faça**
        para $z \leftarrow 1$ **até** p **faça**
            $M \leftarrow (M, M_{i,j,z} \leftarrow A_{z,j}B_{i,z})$
        fim para
        para $z \leftarrow 2$ **até** p **faça**
            $S \leftarrow (S, S_{i,j,z} \leftarrow M_{i,j,z-1} + M_{i,j,z})$
        fim para
    fim para
fim para
retornar M, S

fazê-lo é buscar a melhor reutilização de um objeto de memória por um processador, antes que ele seja transferido para outro espaço e esteja indisponível ou necessite uma nova transferência.

De volta à multiplicação matricial em blocos, e ao levar em conta a questão da reutilização de memória, é desejável que, quando um bloco de matriz $A_{i,j}$ ou $B_{i,j}$ for transferido para um processador para execução de algum M_i em que esteja envolvido, este processador se encarregue de todos os outros M_i que o envolvam. Portanto, quando um processador recebe um bloco, ele na realidade se encarrega de todo um conjunto de operações que envolvem aquele e os outros blocos recebidos conjuntamente.

Dessa forma, configura-se uma dependência entre os blocos das matrizes A e B a serem multiplicadas e as operações de multiplicação em que estão envolvidos. O grafo $G = (V, E)$, ilustrado pela figura 4.1, representa a relação de dependência entre eles, onde o conjunto de vértices V são os blocos e o conjunto de arestas E representam participação conjunta de dois blocos em uma multiplicação M. Por ele, fica claro que é possível separar os blocos em dois grupos que não possuem nenhuma dependência entre si.

Como multiplicações possuem apenas dois fatores, as arestas E representam não somente as dependências entre eles, mas também as multiplicações. Por tanto, define-se as

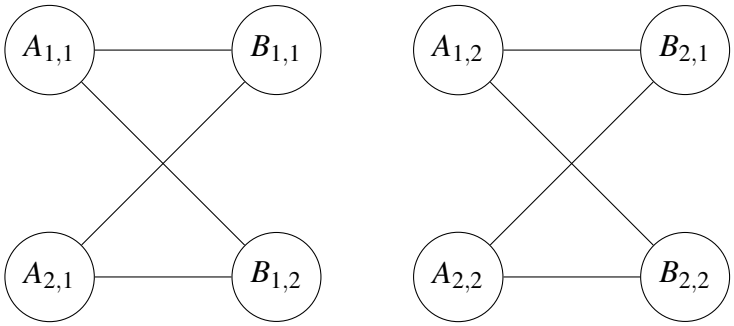


Figura 4.1 – Grafo de dependências entre blocos de matrizes que participam de uma mesma multiplicação.

multiplicações que têm dependências em comum através de um segundo grafo $G_m = (V_m, E_m)$, com vertices $V_m = E$ e arestas $E_m = V$, ilustrado pela figura 4.2. Através dele, conclui-se quais multiplicações M_i devem ser feitas em um mesmo processador devido à reutilização de memória.

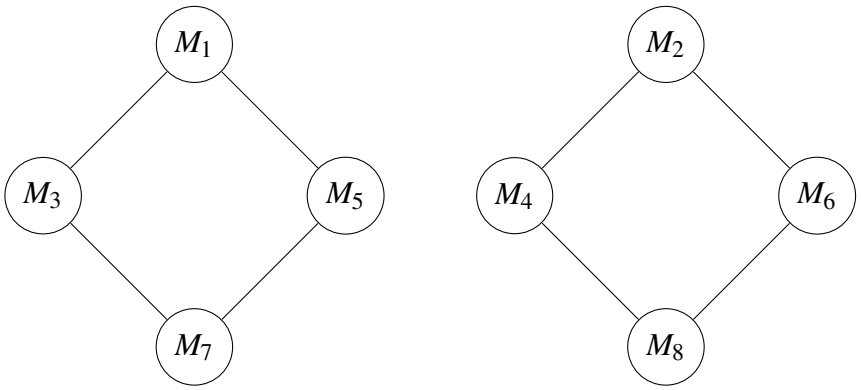


Figura 4.2 – Grafo G_m que mostra as multiplicações que dependem de um mesmo fator.

Adiante, define-se o conjunto $T = \{T_1, T_2, T_3\}$ de tarefas extraídas de uma multiplicação matricial, em que $T_1 = \{M_1, M_3, M_5, M_7\}$, $T_2 = \{M_2, M_4, M_6, M_8\}$ e $T_3 = \{S\}$. Como as somas de T_3 dependem dos resultados das outras duas tarefas, há aí relação de dependência e de hierarquia, já que a primeira não pode ocorrer concorrentemente com as outras. A árvore ilustrada pela figura 4.3 descreve a relação entre as tarefas e a resolução do problema.

4.2.2 Análise de Uso de Memória

Apesar de haver dois tipos de tarefas, S_i e M_i, neste algoritmo, ambas envolvem a mesma quantidade de fatores, de mesma dimensão, como mostra a expressão regular

$$C_{ij} \leftarrow A_{ij}[+|\times]B_{ij}.$$

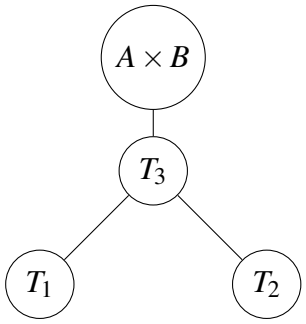


Figura 4.3 – Árvore representando a hierarquia e dependência entre as tarefas extraídas.

Portanto, a quantidade de palavras a ser transferida do hospedeiro para um dispositivo é

$$T_{in} = A_{ij} + B_{ij} = \frac{m \cdot k}{p^2} + \frac{k \cdot n}{p^2} = \frac{1}{p^2}(m \cdot k + k \cdot n),$$

enquanto a quantidade para transferência inversa é

$$T_{out} = C_i = \frac{m \cdot n}{p^2}.$$

A quantidade mínima de memória alocada em um dispositivo para execução de qualquer tarefa T é

$$MA = C_{ij} + B_{ij} + A_{ij} = \frac{1}{p^2}(m \cdot n + m \cdot k + k \cdot n)$$

palavras. Por esse motivo, qualquer que seja o tamanho do problema a ser processado e da memória disponível em um dispositivo do sistema, este pode participar da execução desse problema desde que p seja grande o suficiente, em vista que

$$\lim_{p \to \infty} MA = 0.$$

4.3 Comparações entre os Algoritmos

A Tabela 4.1 apresenta comparação entre os algoritmos desenvolvidos nas seções anteriores, Dec_GEPM e Dec_GEPDOT, e também GEMM, que é a execução direta do núcleo BLAS, sem nenhum tipo de decomposição. Os pontos de comparações são as transferências de hospedeiro para dispositivo T_{in}, transferências de dispositivo para hospedeiro T_{out}, quantidade de memória alocada mínima necessária MA, limite de redução de MA em função de p, quantidade de tarefas geradas em T em função de p, e a existência de qualquer dependência entre as tarefas em T.

As tarefas geradas por Dec_GEPDOT requerem menor transferência T_{in}, o que significa que o núcleo GPGPU inicia sua execução antes, mitigando o efeito de transferências posteriores quando combinado com sobreposição de transferências. Quanto às transferências T_{out}, em Dec_GEPM são menores, porém é uma vantagem de menor relevância,

Tabela 4.1 – Comparação entre os algoritmos desenvolvidos para decomposição de GEMM.

Algoritmo	T_{in}	T_{out}	MA	$limMA$	No. t_i	Deps.
GEMM	$k(m+n)*mn$	mn	$k(m+n)*mn$	MA	1	Não
Dec_GEPM	$\frac{m}{p}(k+n)+kn$	$\frac{m}{p}n$	$\frac{m}{p}(k+n)+kn$	kn	p	Não
Dec_GEPDOT	$\frac{1}{p^2}(mk+kn)$	$\frac{mn}{p^2}$	$\frac{1}{p^2}(mn+mk+kn)$	0	$2p^3-p^2$	Sim

já que em ambos os algoritmos essas transferências devem ocorrer sobrepostas a execução de núcleos GPGPU para tarefas seguintes.

A quantidade de memória alocada mínima de Dec_GEPDOT é outra vantagem, por ser menor que os outros. Além disso, por meio deste algoritmo, é possível reduzir qualquer GEMM de forma a caber na memória de qualquer dispositivo do sistema, como indica a função limite de MA.

O algoritmo Dec_GEPM figura como vantajoso nos quesitos de quantidade e dependências de tarefas. Se por um lado ter apenas uma tarefa GEMM torna impossível a ocupação de todo o sistema disponível, ter muitas tarefas menores gera sobrecarga extra no lançamento de múltiplos núcleos GPGPU. A quantidade de tarefas geradas por Dec_GEPM em função de p cresce linearmente, enquanto a de Dec_GEPDOT em função de p cresce cubicamente, apesar de que para p pequenos como 2 ou 3, a sobrecarga deve ser negligível. Dec_GEPM também não gera tarefas com dependências entre si, o que permite maior liberdade na distribuição e torna desnecessária a sincronização entre elas.

5 ESTUDO DE CASO: AMBIENTE COM UMA GPU

Este capítulo está organizado da seguinte forma: a seção 5.1 descreve o ambiente alvo da implementação e de execução de testes; a seção 5.2 explica as implementações dos algoritmos propostos no capítulo 4; por fim, a seção 5.3 descreve a experimentação, a obtenção de resultados e a discussão dos mesmos.

5.1 Especificações Técnicas do Sistema

Os sistemas com a configuração ideal para este trabalho são de acesso restrito, custam caro e são compartilhados. Além disso, o acesso é realizado apenas por terminal de linha de comando, inclusive para o desenvolvimento e depuração. Portanto, um sistema mais simples e acessível se fez necessário para testes preliminares e para o próprio desenvolvimento.

O sistema utilizado para o desenvolvimento, e também na experimentação, deste trabalho foi uma estação de trabalho, localizada na Universidade Federal do Paraná (UFPR) de Curitiba, habilitada a GPGPU, cujas especificações técnicas são mostradas na Tabela 5.1. O sistema possui duas CPUs funcionando em SMP, portanto, vistas como apenas uma no ponto de vista da programação. Cada CPU possui 6 núcleos independentes, e cada um deles é capaz de executar até 2 *threads* simultaneamente, de forma que o sistema ao todo é capaz de executar até 24 *threads* simultaneamente.

Tabela 5.1 – Configuração do sistema da UFPR, utilizado no desenvolvimento e experimentação.

Tipo	Nome	Notas
CPUs	2× Xeon E5-2620 @ 2,0GHz	384 GFLOPS PD
RAM	32GB DDR3 1333MHz	42,6 GB/s
SO	GNU/Linux Debian	*Kernel* 3.2 x86_64
Graphics Bus	PCIe 2.0	8 GB/s
GPU	Tesla C2075	GFLOPS: 515 PD, 1030 PS
VRAM	6GB GDDR5 @ 1,5GHz	144 GB/s

O sistema possui uma GPU para funções GPGPU, fabricada pela NVIDIA, pertencente à arquitetura Fermi [57], e voltada especificamente para sistemas computacionais científicos. Essa GPU possui capacidade de sobrepor transferências de dados com o hospedeiro e execução de *kernels* GPGPU, como exibido pela Figura 3.3. Porém, ela não possui capacidade de sobrepor duas transferências paralelas, como na Figura 3.4. Por fim, diferentemente das voltadas a consumidores, esta GPU é capaz de manipular números em pontos flutuantes tanto

de PS quanto de Precisão Dupla (PD), e, por este motivo, as rotinas deste trabalho foram implementadas para ambos os tipos de dados.

A interface de programação GPGPU ideal para este ambiente e que foi utilizada neste trabalho é a CUDA, na versão 5.0, desenvolvida pela própria NVIDIA. Além de permitir a exploração de todos os recursos específicos e únicos dessa arquitetura de processadores, somente através da CUDA é possível utilizar a CUBLAS. Esta é a biblioteca com implementação BLAS mais estável, com o melhor desempenho médio entre todas, além de ser completa e possuir todos as rotinas disponíveis. Este trabalho utiliza a CUBLAS na versão 2 para todas as chamadas a núcleos BLAS em GPGPU.

5.2 IMPLEMENTAÇÕES

Para experimentação dos algoritmos propostos no capítulo 4, foram implementadas as rotinas Het_2GPU_GEPM, Het_2GPU_GEPM_SP e Het_2GPU_GEPDOT. Elas foram projetadas especificamente para ambientes com CPUs multi-núcleo e duas GPUs, e utilizaram os *kernels* cublasDgemm e cublasSgemm, providos pela biblioteca CUBLAS, que multiplicam duas matrizes de dupla e simples precisão, respectivamente. As subseções seguintes descrevem os algoritmos em alto nível, abstraindo as chamadas específicas de sistema, pois é possível tê-los implementados para outros sistemas GPGPU.

5.2.1 Het_2GPU_GEPM

O Algoritmo 3 começa por definir como 2 a quantidade de partes em que as matrizes A e C serão separadas, na linha 1, por supor haver duas GPUs no ambiente (uma GPU é simulada neste caso). Em seguida, com uma chamada ao Algoritmo 1, monta a lista de tarefas a serem resolvidas.

Adiante, os passos das linhas 7 a 13 colocam as operações seguintes nos lotes de execução: transferência de dados do hospedeiro para dispositivos, execução do *kernel* em cada dispositivo, e transferência dos resultados de dispositivos para o hospedeiro. Cada dispositivo D_i tem associado a si um único lote de execução, na linha 6. A condicional da linha 9 evita que tempo seja perdido com uma transferência inútil, caso β indique o não acúmulo de C. A execução do algoritmo espera em 15, para que ambos lotes de execução terminem suas tarefas. Em seguida, o resultado já estará pronto em C.

5.2.2 Het_2GPU_GEPM_SP

O Algoritmo 4 é similar ao Algoritmo 3, porém ele visa a utilização de sobreposição de transferências de memória para cada GPU. Isso é feito ao separar as matrizes no dobro de partes, nas linhas 2 e 3, em relação à quantidade de GPUs no sistema, de tal forma a haver duas partes para cada GPU. Dessa forma, uma parte de A que seria transferida em um só passo para algum D_i, passa a exigir dois passos, que podem ser feitos por dois

Algoritmo 3 Het_2GPU_GEPM

Entrada: $A \in \mathbb{R}^{m\times k}$, $B \in \mathbb{R}^{k\times n}$, $C \in \mathbb{R}^{m\times n}$ e $\alpha, \beta \in \mathbb{R}$
Saída: $C \in \mathbb{R}^{m\times n}$

1: $p \leftarrow 2$ /* quantidade de GPUs no sistema */
2: $T \leftarrow Dec\_GEPM(A,B,C,\alpha,\beta,p)$ /* ver algoritmo 1 */
3: Definir $L = (l_1, \cdots, l_p)$, onde cada l_i é um lote de execução
4: **para** $i \leftarrow 1$ **até** p **faça**
5: Definir A_t e C_t como as matrizes envolvidas em t_i
6: Associar l_i a D_i
7: $l_i \leftarrow (l_i, A_t \rightarrow D_i)$
8: $l_i \leftarrow (l_i, B \rightarrow D_i)$
9: **se** $\beta \neq 0$ **então**
10: $l_i \leftarrow (l_i, C_t \rightarrow D_i)$
11: **fim se**
12: $l_i \leftarrow (l_i, t_i)$ /* $t_i = C_i \leftarrow \alpha A_i B + \beta C_i$ */
13: $l_i \leftarrow (l_i, C_t \rightarrow H)$
14: **fim para**
15: Sincronizar L
16: **retornar** C

lotes concorrentemente, abrindo oportunidade para a sobreposição. A mesma coisa acontece na transferência de C de volta das GPUs para a CPU.

Outra mudança em relação ao Algoritmo 3 está na linha 5 que indica que os dados referentes à matriz B são colocados em página fixa da memória, pois eles serão transferidos para dois dispositivos, economizando dessa forma uma transferência entre páginas caso ficasse inteiramente a cargo do sistema. Essa modificação foi feita para medir a diferença de desempenhos entre os dois algoritmos, no quesito transferências de memória de B, do hospedeiro para os dispositivos.

Uma ultima mudança está entre as linhas 6 e 9, que indicam o envio de B para os dispositivos antes do laço principal que agenda as tarefas. Essa modificação é necessária para garantir que as transferências de B não atrapalhem a sobreposição nas transferências de A_i e C_i, já que é impossível controlar a ordem de execução de tarefas dos lotes concorrentes. Como a sobreposição só ocorre quando a execução de um *kernel* for iniciada, o ideal é que logo após a transferência de um A_i, o *kernel* dele inicie execução, para que o outro A_i sobreponha a operação.

5.2.3 Het_2GPU_GEPDOT

Assim como os algoritmos anteriores, o Algoritmo 5 inicia indicando em quantas partes as matrizes serão separadas. Porém, como utiliza o Algoritmo 2 para decomposição matricial, cada matriz é separada em p^2 partes, onde p é a quantidade de GPUs disponíveis no sistema. Após a decomposição de GEMM em multiplicações e somas, na linha 2, o algoritmo separa essas operações em grupos de tarefas de forma que T_{p+1} contenha as

Algoritmo 4 Het_2GPU_GEPM_SP

Entrada: $A \in \mathbb{R}^{m\times k}$, $B \in \mathbb{R}^{k\times n}$, $C \in \mathbb{R}^{m\times n}$ e $\alpha, \beta \in \mathbb{R}$
Saída: $C \in \mathbb{R}^{m\times n}$

1: $d \leftarrow 2$ /* quantidade de GPUs no sistema */
2: $p \leftarrow d \times 2$
3: $T \leftarrow Dec\_GEPM(A,B,C,\alpha,\beta,p)$ /* ver algoritmo 1 */
4: Definir $L = (l_1, \cdots, l_p)$, onde cada l_i é um lote de execução
5: Colocar B em página fixa
6: **para** $i \leftarrow 1$ **até** p **passo** 2 **faça**
7: $l_i \leftarrow (l_i, B \rightarrow D_{i \bmod d})$
8: **fim para**
9: Sincronizar L
10: **para** $i \leftarrow 1$ **até** p **faça**
11: Definir A_t e C_t como as matrizes envolvidas em t_i
12: $l_i \leftarrow (l_i, A_t \rightarrow D_{i \bmod d})$
13: **se** $\beta \neq 0$ **então**
14: $l_i \leftarrow (l_i, C_t \rightarrow D_{i \bmod d})$
15: **fim se**
16: $l_i \leftarrow (l_i, t_i)$ /* $t_i = C_i \leftarrow \alpha A_i B + \beta C_i$ */
17: $l_i \leftarrow (l_i, C_t \rightarrow H)$
18: **fim para**
19: Sincronizar L
20: **retornar** C

somas, e os outros T_i contenham as tarefas destinadas a cada GPU D_i.

Adiante, é definido um lote de execução para cada GPU e sua respectiva lista de tarefas. Nas linhas 16 e 17, o algoritmo agenda a transferência de todos os dados necessários a execução de todo um grupo T_i em D_i. Apesar de inviabilizar a sobreposição de *kernels* e transferências, há ganho de desempenho ao realizar todas as transferências de uma só vez, como explica o manual de programação CUDA (como nota, o manual de OpenCL, da AMD, tem recomendação similar). No próximo laço, o algoritmo agenda a execução dos *kernels* em l_i e um ponto de sincronização E_j. A necessidade desse elemento é justificada nas linhas 24 e 25, que indicam que l_o deve esperar l_i atingir o marcador E_j para que possa prosseguir com sua própria execução. Dessa forma, é utilizada a sobreposição entre as execuções t_j e as transferências $M_t \rightarrow H$, sem correr o risco de l_o executar a operação de transferência antes que M_t esteja terminado.

Ao final do algoritmo, espera-se todas as tarefas de transferência de dispositivos para hospedeiro terminarem, para somente então iniciar a resolução das tarefas de T_{p+1}, que nada mais são que as somas dos resultados advindos de cada GPU, remontando então a matriz C.

Algoritmo 5 Het_2GPU_GEPDOT

Entrada: $A \in \mathbb{R}^{m\times k}$, $B \in \mathbb{R}^{k\times n}$, $C \in \mathbb{R}^{m\times n}$ e $\alpha, \beta \in \mathbb{R}$
Saída: $C \in \mathbb{R}^{m\times n}$

1: $p \leftarrow 2$ /* quantidade de GPUs no sistema */
2: $(M,S) \leftarrow Dec\_GEPDOT(A,B,C,\alpha,\beta,p)$ /* ver algoritmo 2 */
3: Seja $T = (T_1, \cdots, T_{p+1})$, onde cada T_i é um conjunto de tarefas
4: $T_{p+1} \leftarrow S$
5: **para** $i \leftarrow 1$ **até** p **faça**
6: **para** $j \leftarrow 1$ **até** p **faça**
7: **para** $z \leftarrow 1$ **até** p **faça**
8: $T_z \leftarrow (T_z, M_{i,j,z})$
9: **fim para**
10: **fim para**
11: **fim para**
12: Definir $L = (l_1, \cdots, l_p)$, onde cada l_i é um lote de execução
13: **para** $i \leftarrow 1$ **até** p **faça**
14: Definir A_T e B_T como as matrizes envolvidas em T_i
15: Associar l_i a D_i
16: $l_i \leftarrow (l_i, A_T \rightarrow D_i)$
17: $l_i \leftarrow (l_i, B_T \rightarrow D_i)$
18: **para cada** t_j em T_i **faça**
19: Definir M_t como a matriz resultado envolvida em t_j
20: Definir o lote l_o, associado a D_i
21: Definir marcador E_j
22: $l_i \leftarrow (l_i, t_j)$ /* $t_j = M_t \leftarrow \alpha A_t B_t$) */
23: $l_i \leftarrow (l_i, E_j)$
24: $l_o \leftarrow (l_o, SincronizaE_j)$
25: $l_o \leftarrow (l_o, M_t \rightarrow H)$
26: **fim para**
27: **fim para**
28: Sincronizar todos os l_o
29: Realizar todo T_{p+1} em H
30: **retornar** C

5.3 EXPERIMENTAÇÃO

Para a experimentação, foram medidos os tempos das rotinas desenvolvidas nas seções 5.2.1, 5.2.2, 5.2.3 e, para fins de comparação, de uma chamada direta ao *kernel* GEMM da CUBLAS, identificada nesta seção como CUGEMM. Todas essas rotinas citadas possuem variações para PS e PD, para um total de oito rotinas, que foram executadas para matrizes com dimensões $d \times d$, com $d \in \mathbb{N}$ partindo de 500 até 5000, aumentando de 20 em 20. Esse espaço de dimensões matriciais foi escolhido pois parte de um ponto em que o desempenho de GEMM em GPUs é inferior ao de GEMM em CPUs, engloba as dimensões em que eles se equiparam e permite visualizar a acelerada escalabilidade do desempenho de GPGPU à medida que a quantidade de dados aumenta.

Para a tomada de tempos utilizou-se chamadas específicas do sistema operacional com precisão superior a microssegundos, baseadas no contador de ciclos da CPU. Uma chamada a essas rotinas marca o tempo imediatamente antes da rotina a ser medida iniciar e outras chamadas marcam a variação do tempo de acordo com o estágio de execução que se deseja medir. Para cada dimensão matricial testada todas as rotinas foram executadas 101 vezes, e a média dos tempos anotada. Como sistemas operacionais modernos gerenciam vários contextos simultaneamente, a realização de medida de tempo de rotinas está sujeita à interferências ocasionais que distorcem seu tempo real de execução. Portanto, executa-se várias vezes a rotina para que a média dos resultados reflita o tempo comum de uso. A primeira execução é descartada para evitar interferências de criação e inicialização de contextos inerentes à CUDA (o mesmo ocorre em OpenCL), e também interferências que ocorrem quando o sistema carrega bibliotecas dinâmicas para a memória.

Como o sistema da UFPR só possui uma GPU, as rotinas foram adaptadas para verificar essa condição e gerar apenas a parte da carga destinada a uma das GPUs. O desempenho final é inferido simulando a existência de dois dispositivos a partir do desempenho do único dispositivo atuante.

Por fim, através da execução no sistema daUFPR, mediu-se os tempos dos passos que compõem as rotinas, que são três no caso de CUGEMM, Het_2GPU_GEPM e Het_2GPU_GEPM_SP, e quatro no de Het_2GPU_GEPDOT, sendo: a transferência de dados da CPU para a(s) GPU(s); a execução da(s) multiplicação(ões) nela(s); a(s) transferência(s) de volta dos dados para a CPU; e, apenas para Het_2GPU_GEPDOT, a adição pelas CPUs dos blocos de matrizes resultantes para calcular o resultado final.

5.3.1 Resultados

Os tempos totais de execução de cada rotina para cada dimensão das matrizes entradas são exibidos pelas Figuras 5.1 e 5.2. Já as Tabelas 5.2 e 5.3 disponibilizam os tempos tomados em cada passo das rotinas comparadas. Como são milhares de entradas realizadas, é impraticável listar todas. No entanto, os desempenhos de entradas com valores próximos são similares, de forma que é possível escolher apenas algumas e manter uma representação aproximada de todas.

As tabelas estão organizadas da seguinte forma: a coluna “Rotina” indica qual a rotina utilizada para os tempos da linha; a coluna “Dimensões” indica as dimensões m, n e k das matrizes A, B e C envolvidas na multiplicação. A coluna “CPU p/ GPU” indica o tempo gasto na transferência de dados da CPU para a GPU. A coluna “*Kernel*” indica o tempo despendido na multiplicação dos blocos matriciais na GPU. A coluna “GPU p/ CPU” indica o tempo gasto na volta dos resultados da GPU para a CPU. Por fim, a coluna “AXPY” indica o tempo utilizado pela CPU para a adição entre os blocos matriciais advindos da GPU e formação da matriz C resultante. Como esse estágio de execução só existe na rotina Het_2GPU_GEPDOT, apenas as colunas relativas a ele estão preenchidas. As figuras 5.3 a 5.6

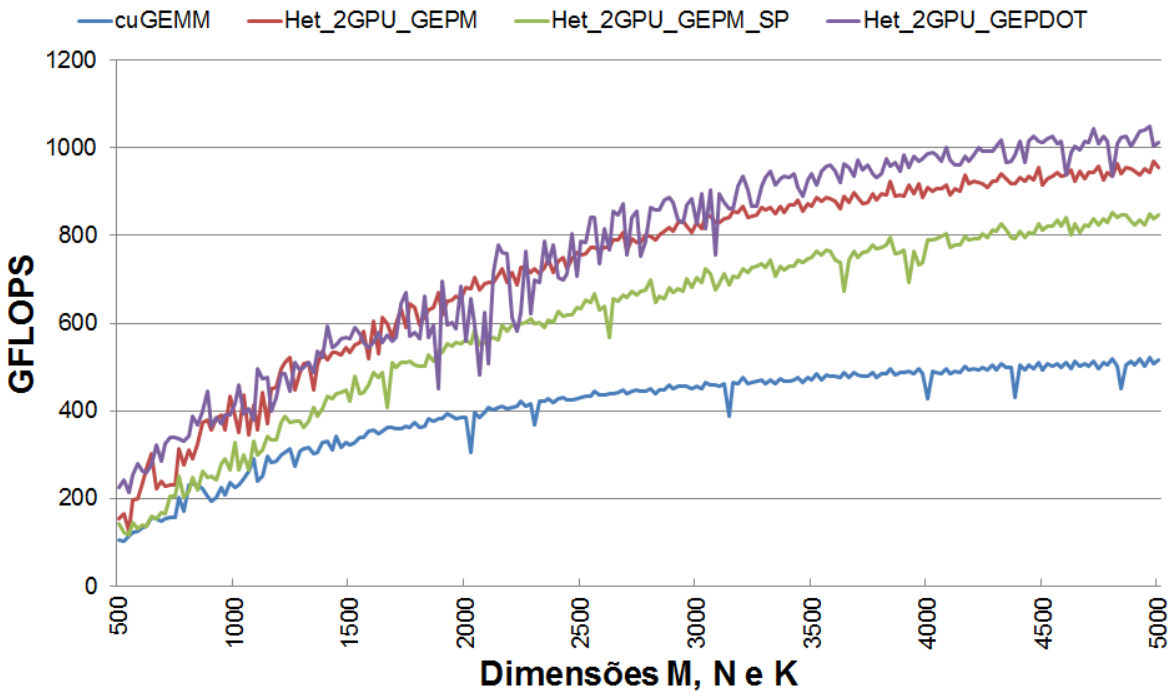


Figura 5.1 – Desempenho das rotinas para precisão simples na UFPR.

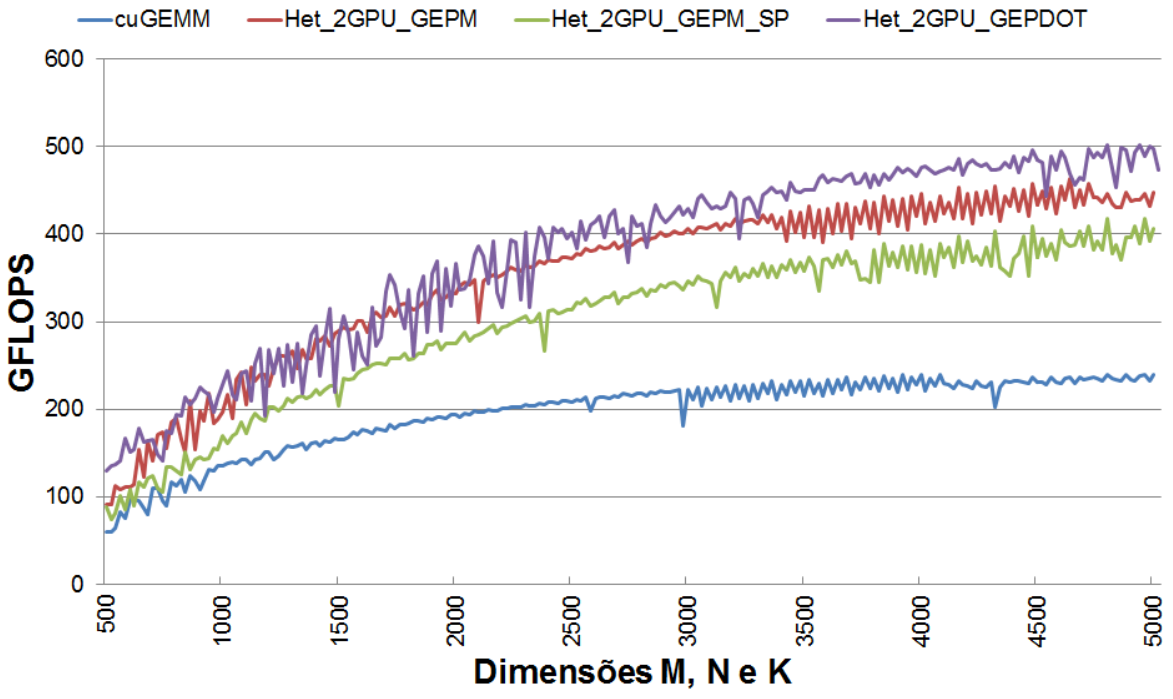


Figura 5.2 – Desempenho das rotinas para precisão dupla na UFPR.

ilustram as medições exibidas na Tabela 5.2, enquanto as figuras 5.7 a 5.10 ilustram as da Tabela 5.3.

Tabela 5.2 – Tempos, em segundos, de cada passo de execução das rotinas experimentadas na UFPR, para **precisão simples**.

Rotina	Dimensões	CPU p/ GPU	*Kernel*	GPU p/ CPU	AXPY
CUGEMM	500	0,0011	0,0006	0,0007	
	1000	0,0035	0,0033	0,0020	
	1500	0,0068	0,0106	0,0037	
	2000	0,0113	0,0246	0,0058	
	2500	0,0161	0,0483	0,0084	
	3000	0,0237	0,0828	0,0121	
	3500	0,0315	0,1342	0,0162	
	4000	0,0669	0,1953	0,0380	
	4500	0,0519	0,2917	0,0266	
	5000	0,0637	0,3885	0,0328	
Het_2GPU_GEPM	500	0,0009	0,0003	0,0005	
	1000	0,0024	0,0018	0,0010	
	1500	0,0052	0,0055	0,0021	
	2000	0,0083	0,0125	0,0028	
	2500	0,0126	0,0242	0,0045	
	3000	0,0178	0,0414	0,0059	
	3500	0,0240	0,0666	0,0084	
	4000	0,0313	0,0993	0,0103	
	4500	0,0394	0,1461	0,0136	
	5000	0,0485	0,1983	0,0161	
Het_2GPU_GEPM_SP	500	0,0005	0,0003	0,0005	
	1000	0,0016	0,0018	0,0011	
	1500	0,0039	0,0055	0,0027	
	2000	0,0061	0,0123	0,0031	
	2500	0,0095	0,0243	0,0051	
	3000	0,0135	0,0438	0,0063	
	3500	0,0180	0,0667	0,0092	
	4000	0,0239	0,1007	0,0108	
	4500	0,0297	0,1466	0,0146	
	5000	0,0372	0,1993	0,0175	
Het_2GPU_GEPDOT	500	0,0003	0,0005	0,0002	0,0001
	1000	0,0017	0,0023	0,0006	0,0001
	1500	0,0044	0,0059	0,0015	0,0002
	2000	0,0068	0,0133	0,0062	0,0038
	2500	0,0092	0,0253	0,0054	0,0003
	3000	0,0127	0,0422	0,0091	0,0030
	3500	0,0163	0,0666	0,0076	0,0004
	4000	0,0211	0,0982	0,0099	0,0005
	4500	0,0267	0,1405	0,0125	0,0006
	5000	0,0334	0,1951	0,0189	0,0007

Tabela 5.3 – Tempos, em segundos, de cada passo de execução das rotinas experimentadas na UFPR, para **precisão dupla**.

Rotina	Dimensões	CPU p/ GPU	*Kernel*	GPU p/ CPU	AXPY
CUGEMM	500	0,0020	0,0009	0,0012	
	1000	0,0051	0,0068	0,0028	
	1500	0,0127	0,0217	0,0065	
	2000	0,0205	0,0516	0,0107	
	2500	0,0320	0,1013	0,0165	
	3000	0,0459	0,1729	0,0237	
	3500	0,0624	0,3024	0,0322	
	4000	0,0822	0,4091	0,0419	
	4500	0,1030	0,6311	0,0530	
	5000	0,1273	0,8503	0,0654	
Het_2GPU_GEPM	500	0,0016	0,0005	0,0007	
	1000	0,0050	0,0035	0,0019	
	1500	0,0090	0,0112	0,0031	
	2000	0,0166	0,0263	0,0055	
	2500	0,0242	0,0515	0,0081	
	3000	0,0348	0,0864	0,0116	
	3500	0,0474	0,1531	0,0158	
	4000	0,0618	0,2041	0,0205	
	4500	0,0780	0,3161	0,0259	
	5000	0,0966	0,4290	0,0319	
Het_2GPU_GEPM_SP	500	0,0008	0,0005	0,0006	
	1000	0,0032	0,0034	0,0017	
	1500	0,0070	0,0112	0,0067	
	2000	0,0123	0,0263	0,0059	
	2500	0,0185	0,0516	0,0088	
	3000	0,0262	0,0887	0,0112	
	3500	0,0358	0,1472	0,0166	
	4000	0,0462	0,2104	0,0206	
	4500	0,0591	0,3357	0,0272	
	5000	0,0715	0,4292	0,0320	
Het_2GPU_GEPDOT	500	0,0010	0,0006	0,0003	0,0001
	1000	0,0036	0,0037	0,0013	0,0002
	1500	0,0068	0,0120	0,0036	0,0030
	2000	0,0108	0,0273	0,0055	0,0003
	2500	0,0168	0,0528	0,0077	0,0004
	3000	0,0243	0,0868	0,0163	0,0006
	3500	0,0326	0,1387	0,0206	0,0008
	4000	0,0421	0,2063	0,0197	0,0010
	4500	0,0528	0,2973	0,0250	0,0013
	5000	0,0654	0,4055	0,0549	0,0012

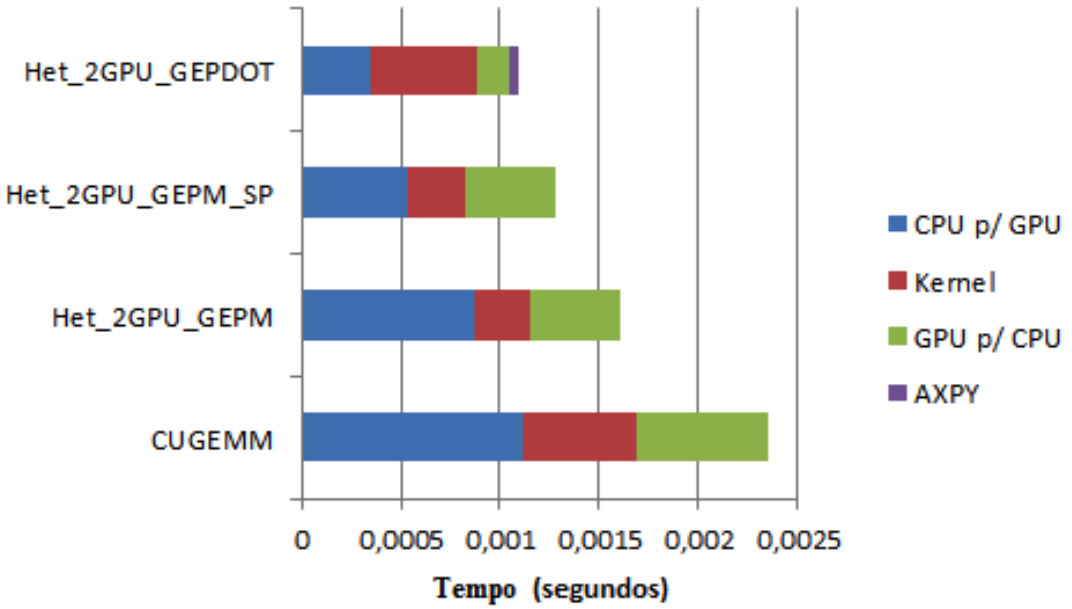


Figura 5.3 – Tempos das rotinas para **precisão simples** e matrizes com dimensões *m*, *n* e *k* igual a **500**.

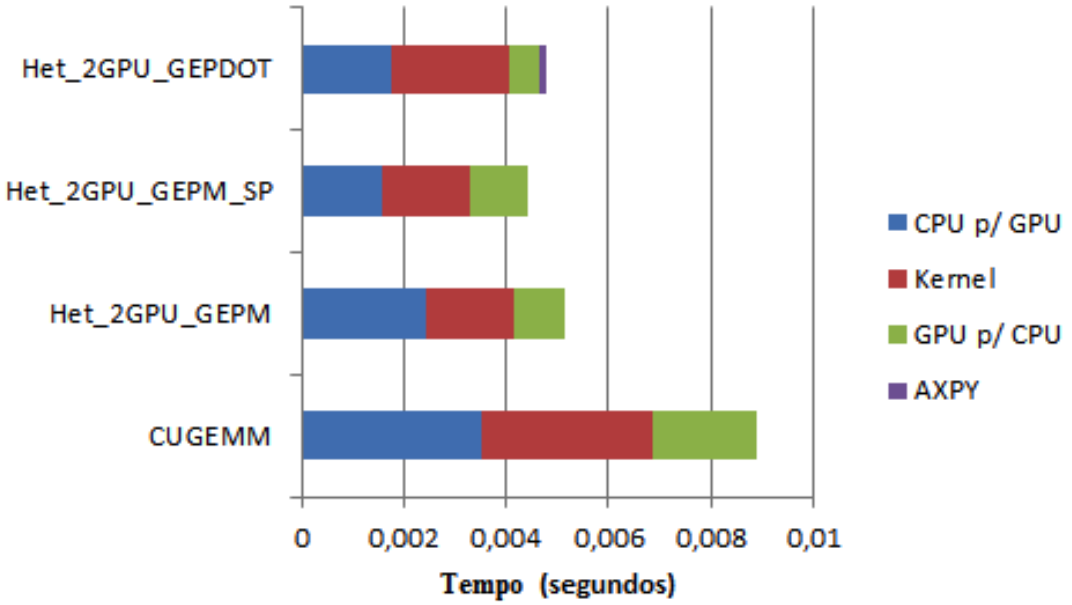


Figura 5.4 – Tempos das rotinas para **precisão simples** e matrizes com dimensões *m*, *n* e *k* igual a **1000**.

5.3.2 Discussão

As figuras 5.1 e 5.2 mostram que o comportamento das rotinas é similar à variância do tipo de dados, PS e PD, quando comparadas umas às outras. Em ambas as figuras, Het_2GPU_GEPDOT inicia com desempenho superior às outras, até que na faixa de dimensões 1000 a 2000, apresenta desempenho inferior à Het_2GPU_GEPM, além de variância no desempenho. A partir da dimensão 2000, o desempenho volta a ser superior e se torna

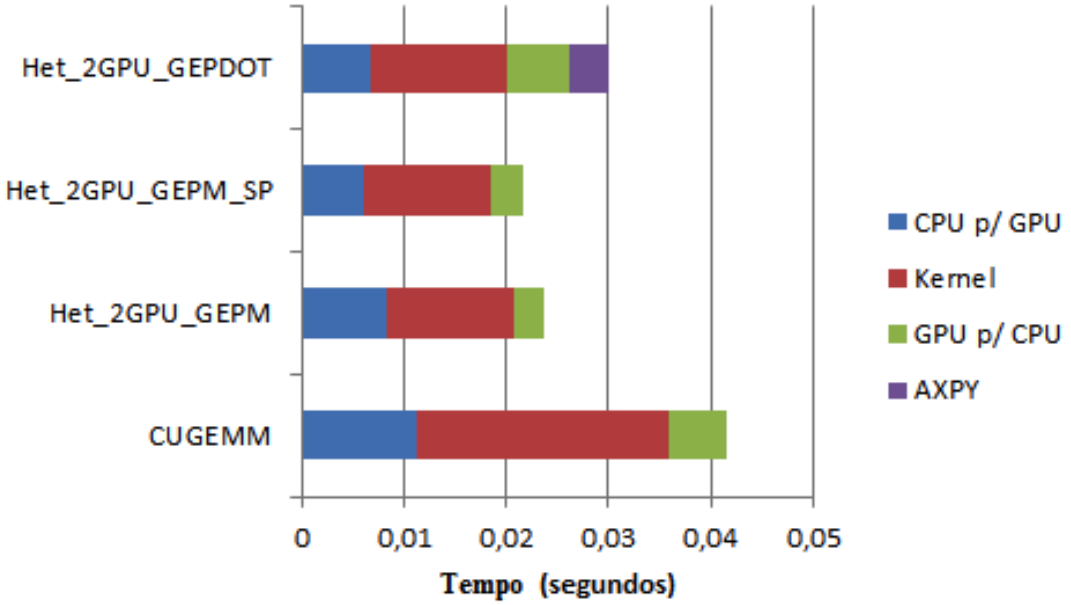


Figura 5.5 – Tempos das rotinas para **precisão simples** e matrizes com dimensões m, n e k igual a **2000**.

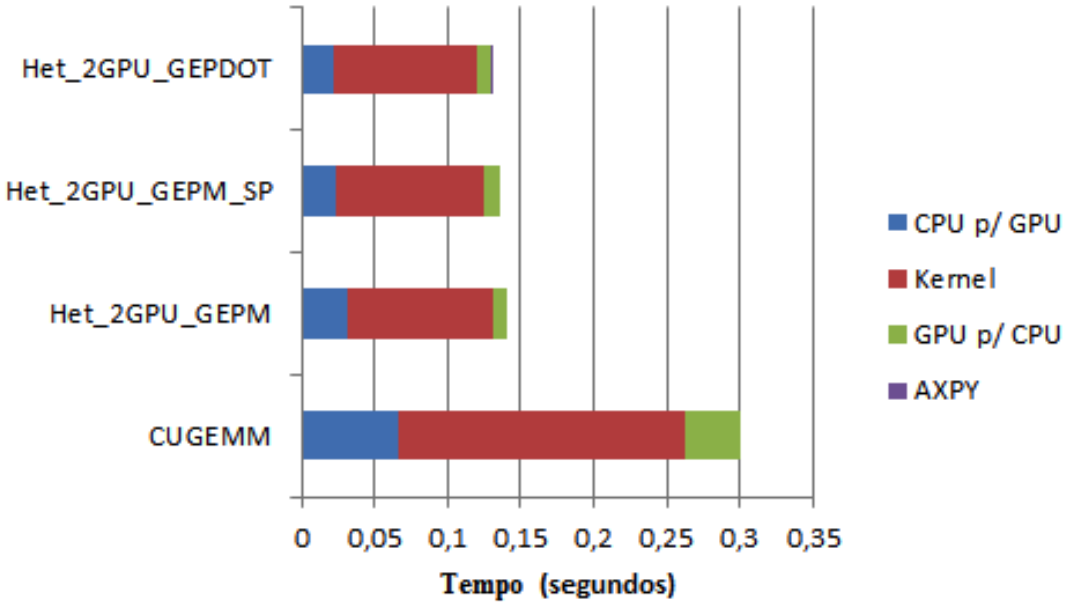


Figura 5.6 – Tempos das rotinas para **precisão simples** e matrizes com dimensões m, n e k igual a **4000**.

consistente, a variância diminui. Esse comportamento de Het_2GPU_GEPDOT mostra que a sobrecarga de gerenciamento para partir, transferir e executar várias tarefas é significativa para matrizes de tamanhos reduzidos. Porém, quando se manipula matrizes de um certo tamanho em diante, o ganho ao ocupar todos os recursos computacionais compensa e resulta em melhor desempenho.

Em relação às variações apresentadas por Het_2GPU_GEPDOT, há dois

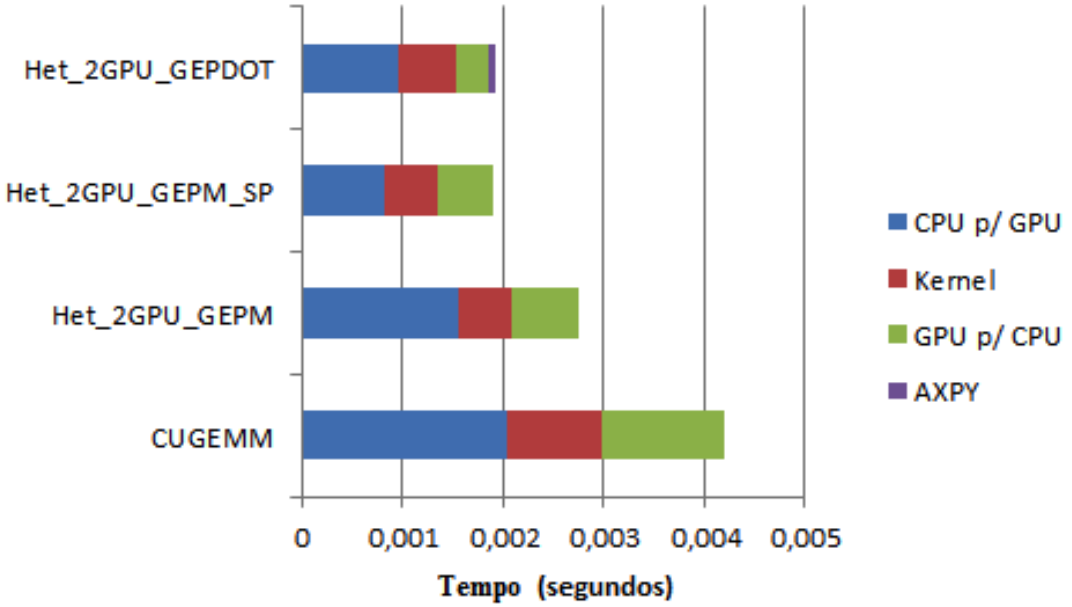


Figura 5.7 – Tempos das rotinas para **precisão dupla** e matrizes com dimensões m, n e k igual a **500**.

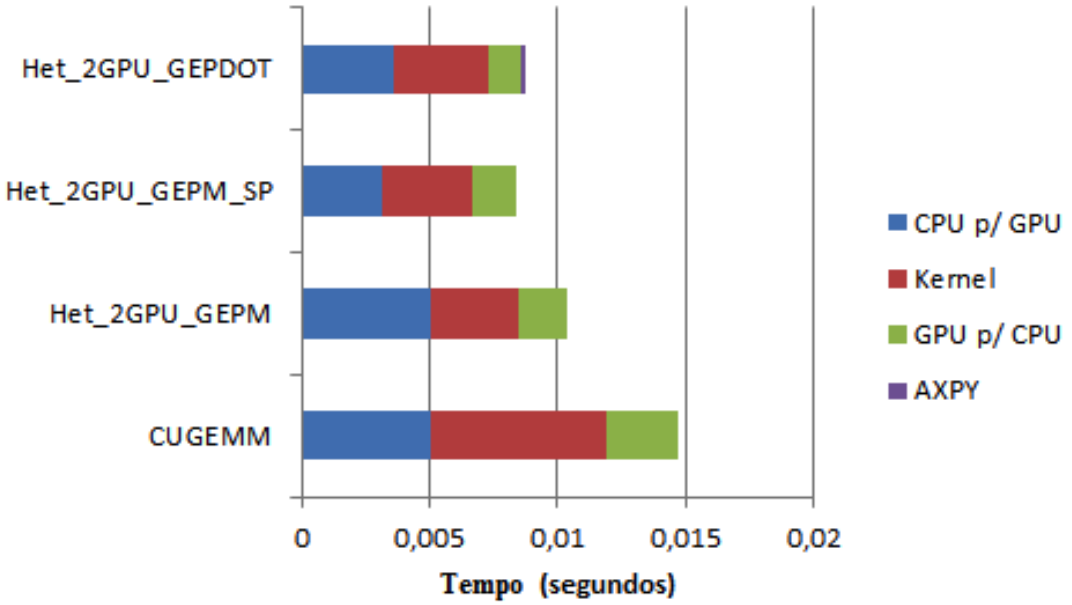


Figura 5.8 – Tempos das rotinas para **precisão dupla** e matrizes com dimensões m, n e k igual a **1000**.

principais motivos. Um deles é a própria sobrecarga para gerenciar a definição e execução de $2p\hat{}3 - p\hat{}2$ tarefas, que aumentam a quantidade de elementos concorrentes e que disputam espaço nas *threads* disponíveis, o que gera comportamento irregular, como se sabe do funcionamento de sistemas com escalonamento dinâmico. Note que este escalonamento ocorre por parte do controlador CUDA, que define qual a próxima tarefa em algum dos vários lotes de execução será executada. O outro motivo está relacionado ao comportamento do núcleo

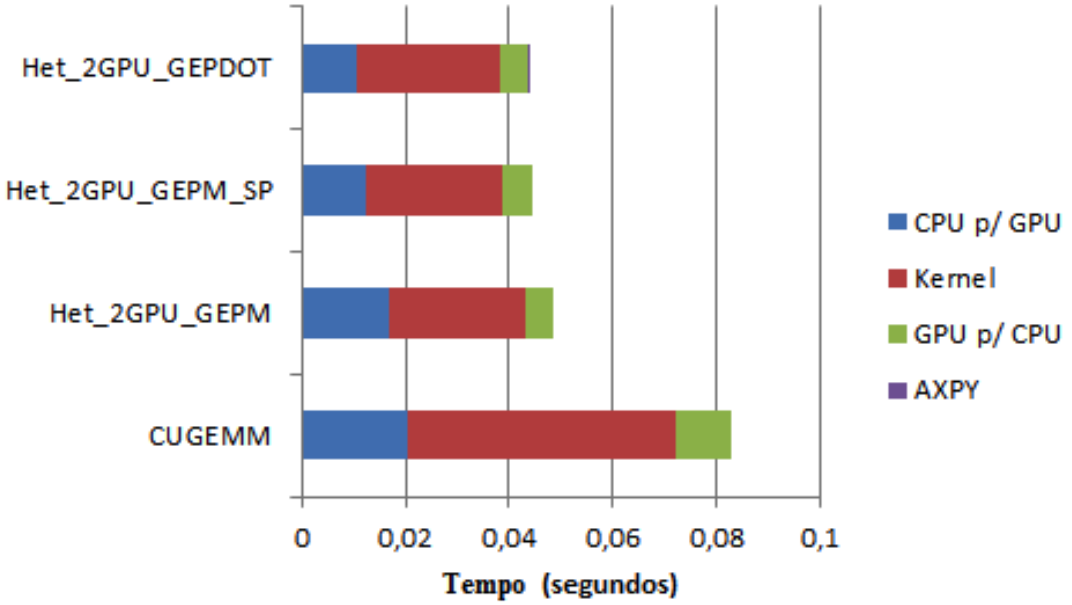


Figura 5.9 – Tempos das rotinas para **precisão dupla** e matrizes com dimensões *m*, *n* e *k* igual a **2000**.

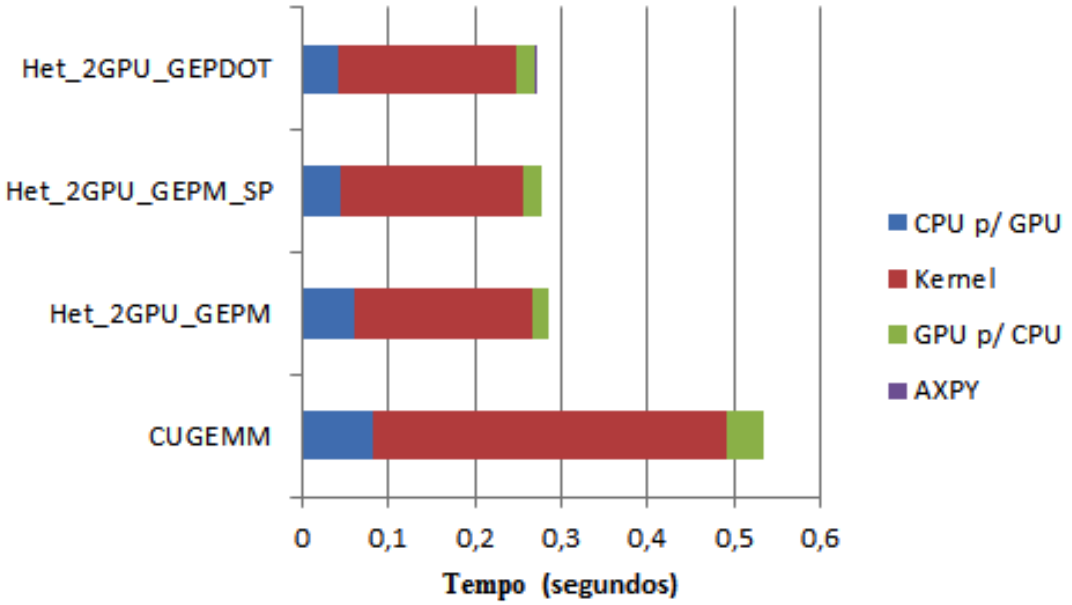


Figura 5.10 – Tempos das rotinas para **precisão dupla** e matrizes com dimensões *m*, *n* e *k* igual a **4000**.

GEMM da CUBLAS, que também não possui desempenho completamente estável. E como Het_2GPU_GEPDOT faz várias chamadas a este núcleo, se as matrizes de entrada forem de dimensão que afetem negativamente o desempenho, então serão várias chamadas de baixo desempenho.

Ainda sobre a rotina Het_2GPU_GEPDOT, ao analisar a influência de um estágio extra de AXPY no hospedeiro pelas figuras 5.3 a 5.6, fica claro que é irrelevante por

ocupar uma fração do tempo total da rotina em todos os casos. Apenas em alguns casos, como o da figura 5.5, o tempo de AXPY conta. Essa ocorrência é independente de qualquer um parâmetros da rotina. Ela está relacionada ao gerenciamento de múltiplos processos por parte do sistema operacional, já que o AXPY utilizado é implementado em OpenMP, que lança e termina as *threads* a cada chamada da função.

Ao analizar as Tabelas 5.2 e 5.3 (e as respectivas figuras), é notado que os tempos dos passos de execução de Het_2GPU_GEPM_SP são menores que os de Het_2GPU_GEPM, ou seja, apresentam melhor desempenho. Porém, as figuras 5.1 e 5.2 contradizem o fato, o desempenho de Het_2GPU_GEPM_SP é inferior em mais de 10%. A explicação está no fato de esta rotina transferir *B* para página fixa da memória antes de realizar sua transferência, e habilitar a sobreposição de transferências e *kernels* GPGPU. O tempo perdido para fixar a matriz não é recuperado ao utilizar os recursos desejados, nem mesmo com o benefício de transferência mais rápida. Fica claro que para compensar a fixação, um conjunto de dados deve ser empregado em várias operações de transferência entre dispositivos e hospedeiro.

Porém, as Tabelas 5.2 e 5.3 confirmam que a rotina Het_2GPU_GEPM_SP compensa para situações em que as matrizes a serem multiplicadas tiverem seus objetos de memória pré alocados em página fixada. Nos casos ilustrados pelas figuras 5.3, 5.4, 5.7 e 5.8, as operações de transferência de dados de hospedeiro para dispositivos tem o tempo reduzido em mais de 40%. E nesses casos de multiplicação entre matrizes de tamanhos reduzidos, as transferências têm influência maior no desempenho final da rotina, como fica claro também na seção 3.1.2.

Por fim, a Tabela 5.4 e as Figuras 5.11 e 5.12 sintetizam os ganhos de desempenho das rotinas desenvolvidas por este trabalho em relação à CUGEMM. A tabela está organizada pelas seguintes colunas: “Precisão” indica o tipo de dados manipulado pela rotina para gerar o resultado indicado na linha; “Dim.” indica as dimensões matriciais; “Het_2GPU_GEPM”, “Het_2GPU_GEPM_SP” e “Het_2GPU_GEPDOT” indicam que os dados apresentados nestas colunas provêm da razão entre a rotina indicada e a rotina CUGEMM.

Os dados da tabela mostram que todas as rotinas desenvolvidas devem apresentam ganho imediato de desempenho, por utilizar duas GPUs, independente dos custos de gerenciamento de tarefas envolvidos. Porém, apenas a rotina Het_2GPU_GEPDOT se mantém o mais próxima de ganho de duas vezes, quando não superior. Isso se deve ao fato de ela possuir maior capacidade de manter elevada a taxa de ocupação dos recursos disponíveis no sistema, que são as *threads* de CPU, as GPUs e o barramento de conexão entre hospedeiro e dispositivos.

A rotina Het_2GPU_GEPM também apresenta bom desempenho e até supera a marca de duas vezes o desempenho de CUGEMM para alguns casos, como em dimensões 4000 e PS. A primeira vista pode parecer anormal ou impossível que isso ocorra, mas o fato é que a execução de CUGEMM para matrizes 4000×4000 pode ser um dos casos ruins para

Tabela 5.4 – Razão entre os desempenhos das rotinas e os de CUGEMM na UFPR.

Precisão	Dim.	Het_2GPU_GEPM	Het_2GPU_GEPM_SP	Het_2GPU_GEPDOT
PS	500	1,46	1,34	2,14
	1000	1,73	1,45	1,86
	2000	1,77	1,46	1,45
	4000	2,13	1,85	2,31
PD	500	1,52	1,50	2,18
	1000	1,56	1,17	1,76
	2000	1,71	1,43	1,90
	4000	1,86	1,62	1,98

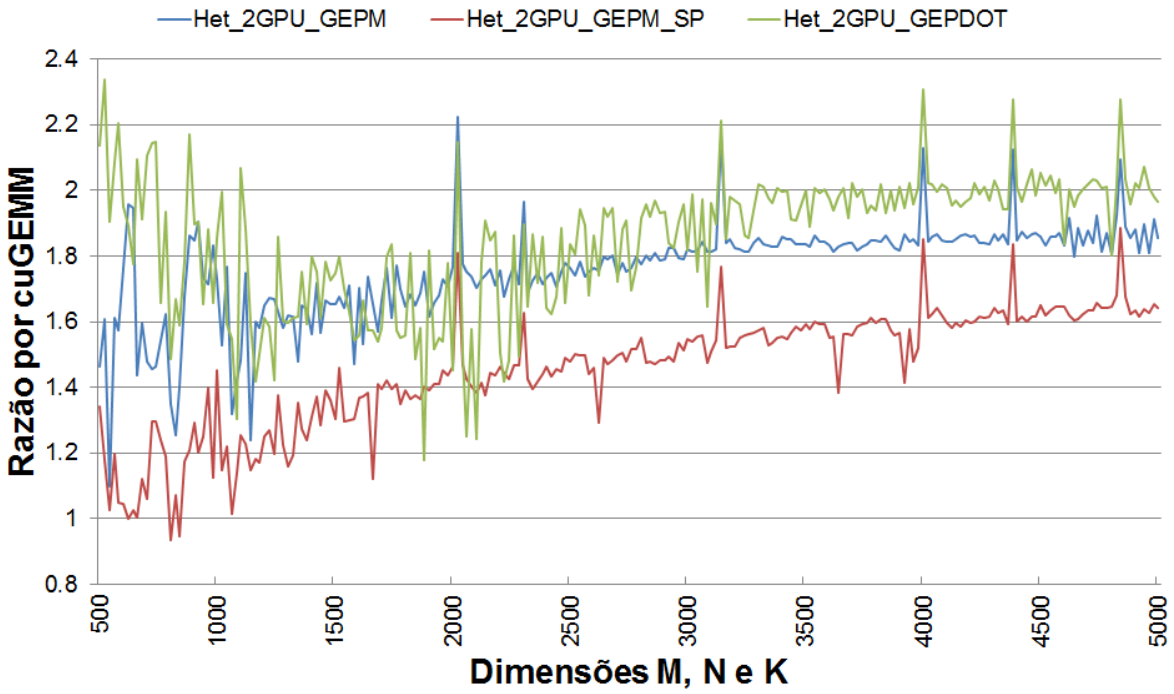


Figura 5.11 – Razão entre os desempenhos das rotinas e os de CUGEMM na UFPR, para precisão simples.

o *kernel* GPGPU, enquanto Het_2GPU_GEPM, nesta mesma situação, efetivamente executa duas multiplicações entre matrizes 2000×4000, que pode ser uma entre as boas combinações para a execução na GPU.

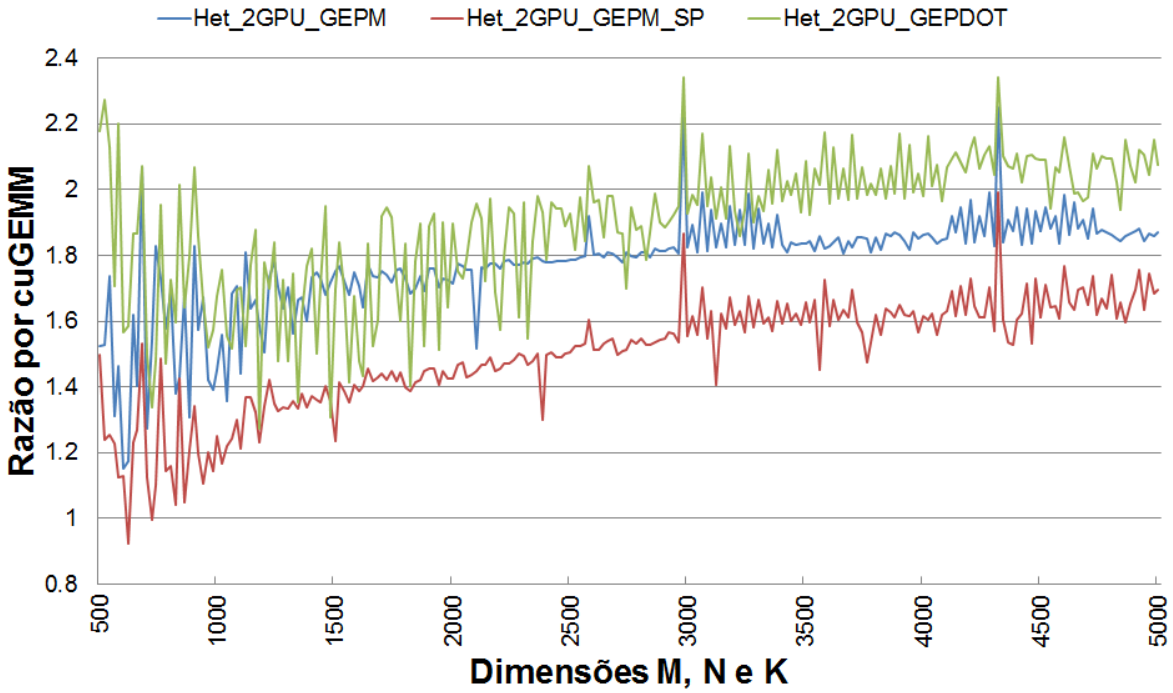


Figura 5.12 – Razão entre os desempenhos das rotinas e os de CUGEMM na UFPR, para precisão dupla.

6 ESTUDO DE CASO: AMBIENTE MULTI-GPU

Este capítulo está organizado da seguinte forma: a seção 6.1 descreve o ambiente alvo da implementação e de execução de testes; a seção 6.2 explica as implementações dos algoritmos propostos no capítulo 4; por fim, a seção 6.3 descreve a experimentação, a obtenção de resultados e a discussão dos mesmos.

6.1 Especificações Técnicas do Sistema

O sistema utilizado para testes foi um nó de *cluster* habilitado à GPGPU do Amazon *Elastic Compute Cloud* (EC2) , identificado por "cg1.x4large", cuja especificação técnica é descrita na Tabela 6.1. O sistema possui duas CPUs funcionando em SMP, portanto, vistas como apenas uma no ponto de vista da programação, e possuem 4 núcleos cada, com capacidade total para executar 16 *threads* simultaneamente. Estão disponíveis também duas GPUs da arquitetura Fermi, similares à do sistema da UFPR, tendo como diferenças relevantes a quantidade de memória de cada dispositivo e a capacidade de sobrepor duas transferências de dados, como na Figura 3.4.

Tabela 6.1 – Configuração do sistema EC2, utilizado na experimentação.

Tipo	Nome	Notas
CPUs	2× Xeon X5570 @ 2,93GHz	–
RAM	24GB DDR3 1333MHz	32 GB/s
SO	GNU/Linux Amazon AMI	*Kernel* 3.2 x86_64
Graphics Bus	PCIe 2.0	8 GB/s
GPU	2× Tesla M2050	GFLOPS: 515 PD, 1030 PS
VRAM	3GB GDDR5 @ 1,5GHz	148,4 GB/s

6.2 Implementações

As rotinas implementadas e utilizadas são as mesmas do estudo de caso do capítulo 5, porém com a diferença de que, neste caso, são utilizadas duas GPUs de fato, sem a necessidade de simular execução e resultados. Para utilizar os dois dispositivos corretamente por meio da CUDA, é necessário que cada rotina inicie duas *threads*, cada uma dedicada a controlar uma única GPU. As tarefas são extraídas normalmente, por apenas uma *thread*, e em seguida são repassadas às *threads* de controle para que as executem. A API de controle de *threads* utilizada foi a OpenMP.

6.3 EXPERIMENTAÇÃO

Para a experimentação, foi feito o mesmo procedimento descrito na seção 5.3. No entanto, como há duas GPUs para receberem, executarem e transferirem de volta dados e tarefas, e como elas funcionam concorrentemente, a medição dos tempos de cada passo das rotinas não é trivial. Isso se deve ao fato de que mesmo para o caso em que uma mesma tarefa é despachada para cada um dos dispositivos, não há garantia de que começarão a execução ao mesmo tempo, e que também terminarão juntas. A solução tomada foi a de medir apenas os passos de execução das rotinas de apenas uma das GPUs.

Uma diferença na medição dos tempos neste estudo de caso está no fato de que, para a rotina Het_2GPU_GEPM_SP, o tempo de sobrecarga oriundo da operação de fixação de página dos dados da matriz B foi retirado do tempo do desempenho total. Ou seja, Het_2GPU_GEPM_SP utilizará os dados de B em página fixa, mas não considerará o tempo necessário para estabelecer essa condição. Como já se sabe, através do estudo de caso do capítulo 5, que, para uma única execução de rotina de multiplicação matricial, não compensa fixar os dados, a ideia é averiguar como é o desempenho para o caso em que tais dados já estejam fixados. Se for vantajoso, um escalonador inteligente pode verificar essa condição e escolher adequadamente entre Het_2GPU_GEPM_SP e Het_2GPU_GEPM.

6.3.1 Resultados

Os tempos totais de execução de cada rotina para cada dimensão das matrizes entradas são exibidos pelas Figuras 6.1 e 6.2, enquanto as Tabelas 6.2 e 6.3 disponibilizam os tempos de execução total das rotinas comparadas. Assim como na seção 5.3.1, são escolhidos apenas alguns dados para figurar nas tabelas. As tabelas estão organizadas da seguinte forma: a coluna "Dim." indica as dimensões m, n e k das matrizes A, B e C envolvidas na multiplicação; e as colunas seguintes indicam o tempo de execução, em segundos, para as respectivas rotinas.

Tabela 6.2 – Tempos, em segundos, de execução das rotinas experimentadas no EC2, para **precisão simples**.

Dim.	CUGEMM	Het_2GPU_GEPM	Het_2GPU_GEPM_SP	Het_2GPU_GEPDOT
500	0,0017	0,0014	0,0028	0,0025
1000	0,0070	0,0046	0,0077	0,0057
1500	0,0186	0,0119	0,0134	0,0125
2000	0,0443	0,0228	0,0225	0,0227
2500	0,0709	0,0402	0,0433	0,0396
3000	0,1121	0,0639	0,0656	0,0611
3500	0,1730	0,0967	0,0935	0,0920
4000	0,2477	0,1365	0,1359	0,1296
4500	0,3518	0,1924	0,1873	0,1812
5000	0,4686	0,2559	0,2495	0,2408

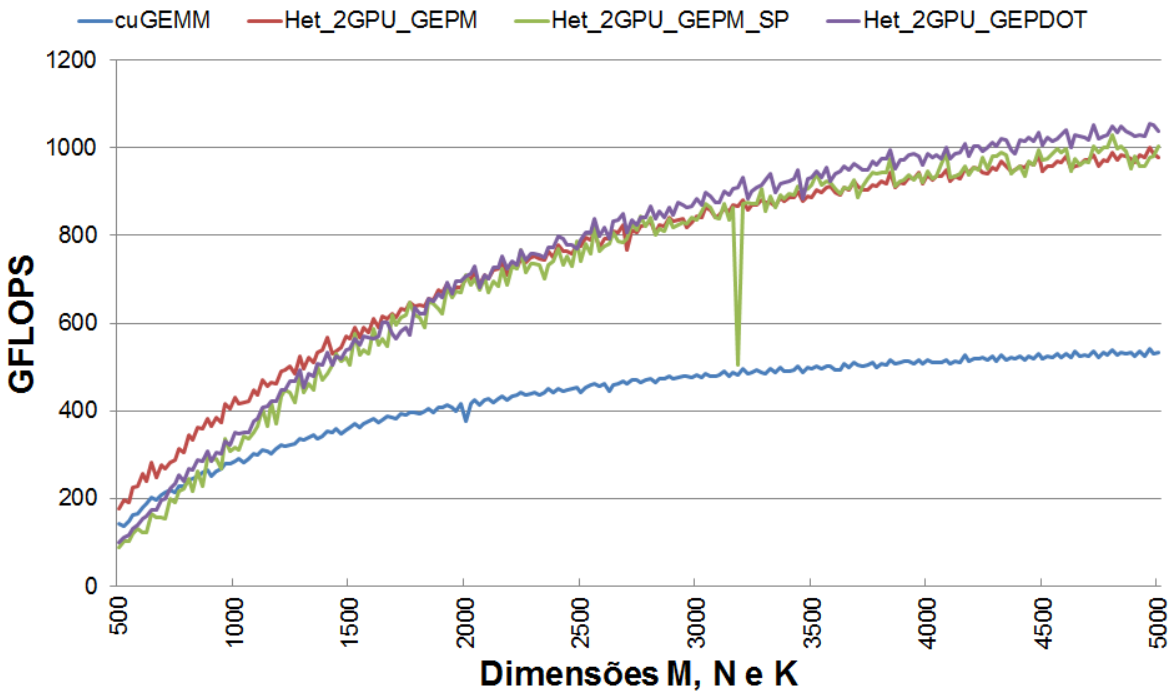


Figura 6.1 – Desempenho das rotinas para precisão simples no EC2.

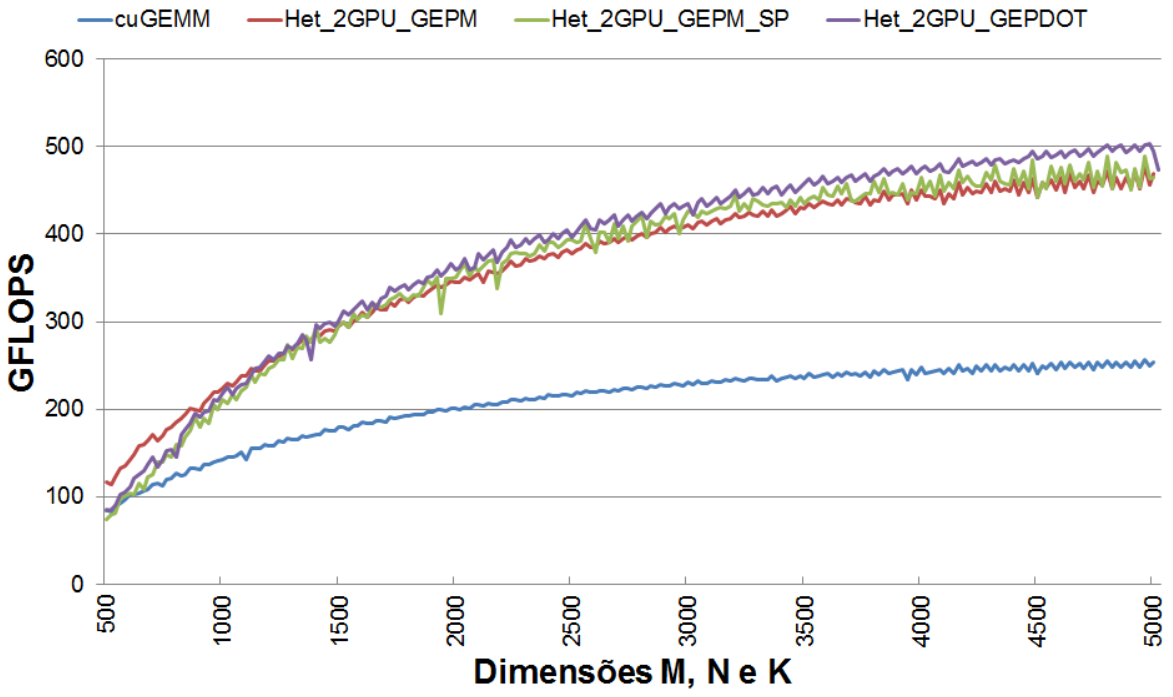


Figura 6.2 – Desempenho das rotinas para precisão dupla no EC2.

6.3.2 Discussão

Os dados sobre desempenho extraídos por execução das rotinas neste sistema com duas GPUs, quando comparados com os do capítulo 5, mostram que apesar de haver diferenças, as tendências são mantidas, o que confere certa validade à simulação como estudo de

Tabela 6.3 – Tempos, em segundos, de execução das rotinas experimentadas no EC2, para **precisão dupla**.

Dim.	CUGEMM	Het_2GPU_GEPM	Het_2GPU_GEPM_SP	Het_2GPU_GEPDOT
500	0,0029	0,0021	0,0034	0,0030
1000	0,0141	0,0089	0,0095	0,0092
1500	0,0377	0,0229	0,0232	0,0224
2000	0,0795	0,0464	0,0466	0,0446
2500	0,1448	0,0827	0,0793	0,0789
3000	0,2344	0,1315	0,1265	0,1240
3500	0,3637	0,1995	0,1973	0,1872
4000	0,5171	0,2843	0,2751	0,2693
4500	0,7583	0,4128	0,4129	0,3751
5000	0,9846	0,5330	0,5381	0,5048

caso. Dentre as igualdades, Het_2GPU_GEPDOT mantém o melhor desempenho para matrizes de tamanho superiores a dimensões 2000, e também mantém a melhor escalabilidade, que são os principais atributos avaliados por este trabalho.

Quanto às diferenças, as Figuras 6.1 e 6.2 mostram que Het_2GPU_GEPDOT tem melhor estabilidade de desempenho em relação às outras, o contrário do apresentado em 5.1 e 5.2, o que é uma vantagem. Uma diferença negativa é o fato de que para dimensões matriciais menores que 2000 para PS e 1500 para PD, Het_2GPU_GEPDOT tem desempenho inferior a Het_2GPU_GEPM. Sobre o melhor desempenho geral de Het_2GPU_GEPM_SP em relação a Het_2GPU_GEPM, isso se deve ao fato de a primeira utilizar os dados de B em página fixa, mas não contabilizar o tempo gasto para fixá-los.

A Tabela 6.4 e as Figuras 6.3 e 6.4, exibem o ganho de desempenho das rotinas desenvolvidas por este trabalho em relação a rotina CUGEMM. Diferentemente do que é apresentado pelas Figuras 5.11 e 5.12, ganhos superiores a duas vezes são improváveis. Mas ainda assim, a rotina Het_2GPU_GEPDOT é a que se mantém mais próxima a esse índice, principalmente para dimensões matriciais superiores a 3500, fato que confirma a capacidade de escalabilidade do algoritmo em relação a quantidade de processadores.

Quanto à rotina Het_2GPU_GEPM, as figuras confirmam que tem o melhor desempenho para dimensões matriciais menores que 2000 para PS e 1500 para PD. No entanto, o algoritmo não apresenta boa escalabilidade em relação a quantidade de processadores disponíveis no sistema, o crescimento do desempenho se mantém praticamente desacelerado a partir de dimensões matriciais 3000.

Tabela 6.4 – Razão entre os desempenhos das rotinas e os de CUGEMM no EC2.

Precisão	Dim.	Het_2GPU_GEPM	Het_2GPU_GEPM_SP	Het_2GPU_GEPDOT
PS	500	1,25	0,62	0,71
	1000	1,52	1,11	1,24
	2000	1,87	1,89	1,88
	4000	1,81	1,83	1,91
PD	500	1,37	0,87	0,99
	1000	1,58	1,49	1,53
	2000	1,71	1,74	1,79
	4000	1,82	1,88	1,92

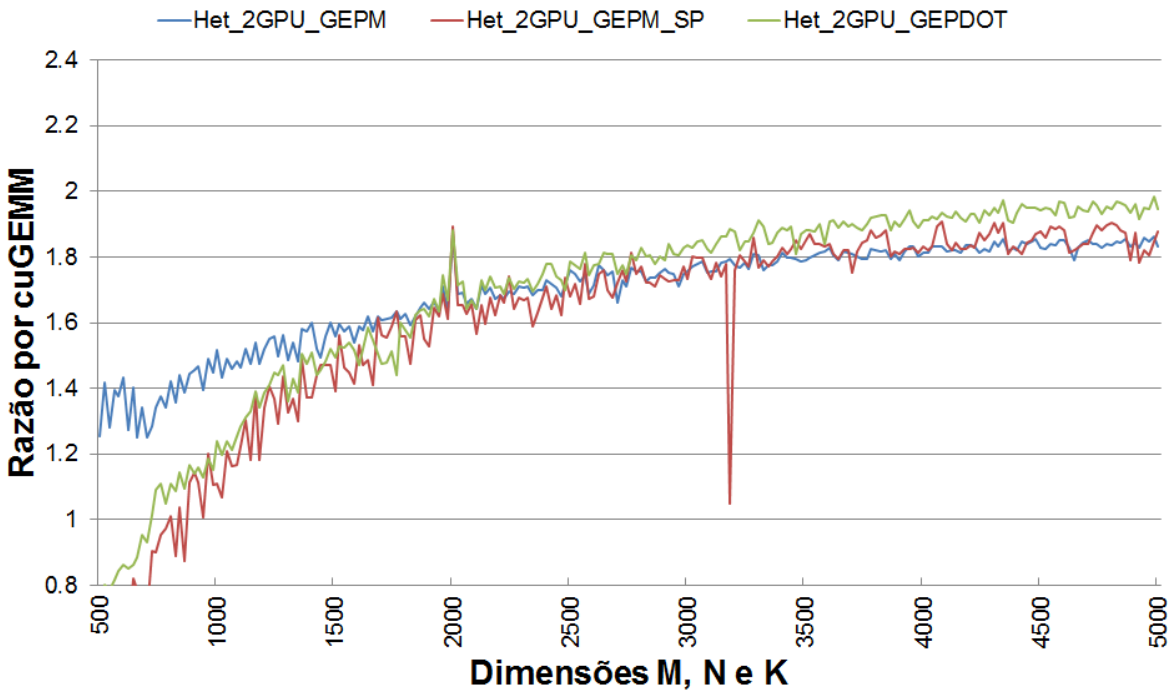


Figura 6.3 – Razão entre os desempenhos das rotinas e os de CUGEMM no EC2, para precisão simples.

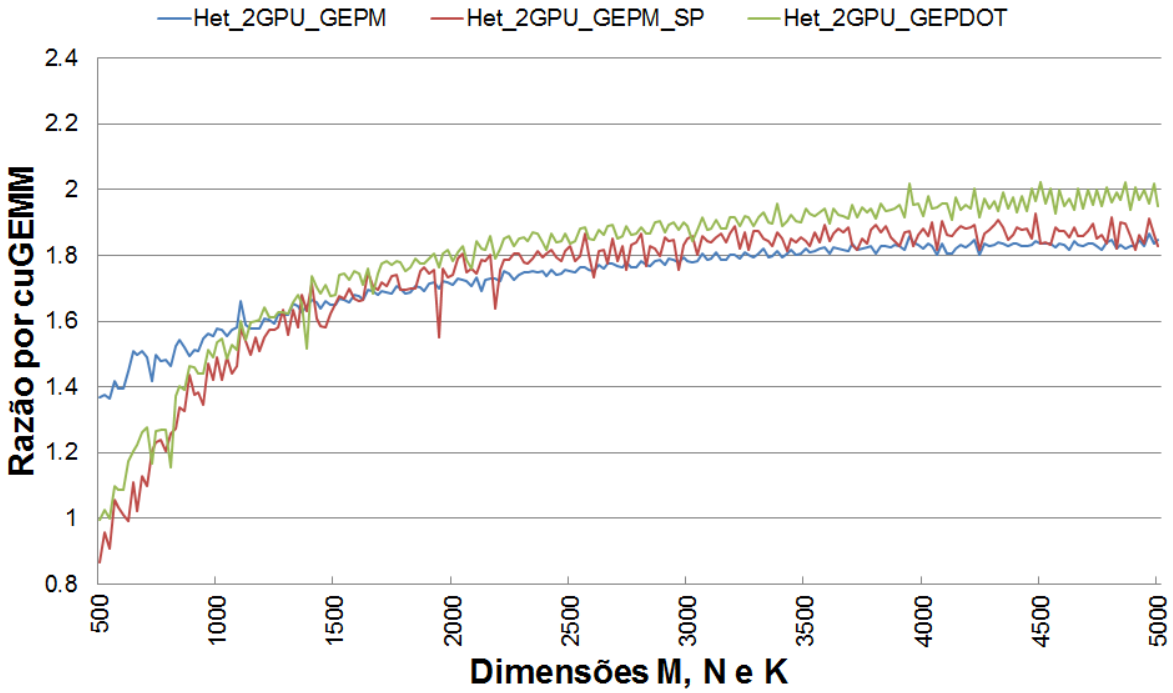


Figura 6.4 – Razão entre os desempenhos das rotinas e os de CUGEMM no EC2, para precisão dupla.

7 CONCLUSÕES

No trabalho foram desenvolvidos os algoritmos Dec_GEPM e Dec_GEPDOT para decomposição de GEMM em tarefas menores, adequadas à distribuição entre os possíveis processadores existentes em sistemas heterogêneos. Dec_GEPM gera tarefas de multiplicações matriciais menores, independentes entre si, mas com redundância de dados nas transferências entre hospedeiro e dispositivos. Foi mostrado que esse algoritmo também pode ser utilizado para explorar recursos de GPUs como a sobreposição de operações de transferências e execução de *kernels*.

Já o algoritmo Dec_GEPDOT decompõe a multiplicação matricial em três grupos de tarefas, sendo dois de multiplicações menores e um de somas vetoriais. Foi demonstrado que apesar de essas tarefas possuírem independência apenas parcial, podem ser organizadas de tal forma a serem resolvidas concorrentemente em processadores distintos que compõem sistemas heterogêneos, como as GPUs. O método também prevê a viabilidade de redução nas transferências de memória entre os processadores e evita a redundância nas transferências de dados. Ele foi projetado de tal forma a distribuir as tarefas de multiplicações, mais pesadas, entre as GPUs e as tarefas de somas para a CPU, normalmente um processador com menor poder computacional.

Os algoritmos desenvolvidos foram utilizados na implementação de rotinas, que tiveram os seus desempenhos medidos. A partir dessa experimentação, foi concluído que ambos os algoritmos se beneficiam de recursos existentes – a disponibilidade de múltiplos e distintos processadores e sobra de banda do barramento de dados – e que não são empregados por núcleos GEMM de outras bibliotecas. O algoritmo Dec_GEPM mostrou melhor aproveitamento dos recursos para multiplicações entre matrizes com dimensões entre 500 a 2000, enquanto Dec_GEPDOT tem melhor aproveitamento a partir de 2000.

Dessa forma, foi possível diminuir o tempo de execução global – que envolve as transferências de dados e a execução do *kernel* GPGPU – de multiplicação entre matrizes, em um ambiente com duas GPUs e uma CPU. O ganho de desempenho varia de acordo com as dimensões matriciais envolvidas, o tipo de dados das matrizes e o algoritmo de decomposição empregado, mas tem-se que o ganho mínimo é em torno de 50% do desempenho de execução em uma única GPU diretamente, para os casos de dimensões matriciais a partir de 1000. E o ganho máximo ultrapassa os 100%, como mostra a execução da rotina Het_2GPU_GEPDOT para matrizes com dimensões acima de 4000.

Por conta disso, este trabalho conclui que os algoritmos desenvolvidos têm capacidade para prover melhor desempenho em ambientes heterogêneos baseados em GPGPU, têm capacidade para fácil escalabilidade de desempenho à medida em que se adiciona mais GPUs no sistema e que se aumenta a banda do barramento de interconexão entre hospedeiros

e dispositivos. Conclui também que, por atingir índices de aumento de desempenho em quase 100% para a maioria dos casos ao dobrar a quantidade de GPUs utilizadas no processo, o trabalho mantém a competitividade com as ferramentas já existentes.

7.1 Trabalhos Futuros

O próximo passo deste trabalho é a construção de um sistema de execução de operações matriciais baseado em fila de tarefas e escalonamento, e a consequente adaptação dos algoritmos por este apresentados ao referido modelo. Dessa forma poderá ser testado em condições semelhantes a de sistemas de produção, além da possibilidade de novas otimizações com o objetivo de aumentar a ocupação total do sistema.

Outra possibilidade seria buscar maneiras de evitar que alguns blocos matriciais sejam transferidos para as GPUs, relegando trabalho para a CPU. Essa situação seria ideal para casos em que a CPU esteja parada, esperando a computação matricial pelos outros processadores terminar, e é uma solução que combina com sistemas de escalonamento dinâmico, e portanto, com sistemas de produção.

A partir do momento que se comprove a viabilidade e utilidade de GEMM adaptado aos ambientes supracitados, será possível avançar para a implementação de outros núcleos BLAS e rotinas LAPACK, seguindo os moldes apresentados neste trabalho.

Referências

[1] IGUAL, F. D. et al. The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations. *Journal of Parallel and Distributed Computing*, 2012, v. 72, n. 9, p. 1134–1143, set. 2012.

[2] LANGDON, W. B. Performing with cuda. In: *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*. New York, NY, USA: ACM, 2011. (GECCO '11), p. 423–430.

[3] NVIDIA. *Compute Unified Device Architecture (CUDA)*. Disponível em: < `www.nvidia.com/object/cuda_home_new.html` >. Acesso em: 21 set. 2012.

[4] KHRONOS. *Open Computing Language (OpenCL)*. Disponível em: < `www.khronos.org/opencl` >. Acesso em: 21 set. 2012.

[5] OHSHIMA, S. et al. Parallel processing of matrix multiplication in a CPU and GPU heterogeneous environment. In: *Proceedings of the 7th international conference on High performance computing for computational science*. Berlin, Heidelberg: Springer-Verlag, 2007. (VECPAR'06), p. 305–318.

[6] VOLKOV, V.; DEMMEL, J. W. Benchmarking GPUs to tune dense linear algebra. In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008. (SC '08), p. 31:1—-31:11.

[7] TOMOV, S.; DONGARRA, J.; BABOULIN, M. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 2010, v. 36, n. 5-6, p. 232–240, jun. 2010.

[8] LAPACK. *Linear Algebra PACKage*. Disponível em: < `www.netlib.org/lapack` >. Acesso em: 21 set. 2012.

[9] BLAS. *Basic Linear Algebra Subprogram*. Disponível em: < `www.netlib.org/blas` >. Acesso em: 21 set. 2012.

[10] GOLUB, G. H.; LOAN, C. F. van. *Matrix Computations*. 3. ed. [S.l.]: The John Hopkins University Press, 1996.

[11] INTEL. *Math Kernel Library (MKL)*. Disponível em: < `software.intel.com/en-us/intel-mkl` >. Acesso em: 21 set. 2012.

[12] ATLAS. *Automatically Tuned Linear Algebra Software*. Disponível em: < `math-atlas.sourceforge.net` >. Acesso em: 21 set. 2012.

[13] GOTO, K. *gotoBLAS*. Disponível em: < www.tacc.utexas.edu/resources/software >. Acesso em: 21 set. 2012.

[14] NVIDIA. *CUDA Basic Linear Algebra Subroutines (CUBLAS)*. Disponível em: < developer.nvidia.com/cuda/cublas >. Acesso em: 21 set. 2012.

[15] FLAME. *Formal Linear Algebra Method*. Disponível em: < z.cs.utexas.edu/wiki/flame.wiki/FrontPage >. Acesso em: 21 set. 2012.

[16] AMD. *AMD Core Math Library (ACML)*. Disponível em: < developer.amd.com/tools/cpu/acml >. Acesso em: 21 set. 2012.

[17] DONGARRA, J. Oral history interview by thomas haigh. *Society for Industrial and Applied Mathematics*, 2004.

[18] EISPACK. *EISPACK*. Disponível em: < www.netlib.org/eispack >. Acesso em: 21 set. 2012.

[19] LINPACK. *LINPACK*. Disponível em: < www.netlib.org/linpack >. Acesso em: 21 set. 2012.

[20] TOP500. *TOP500 Supercomputers Site*. Disponível em: < http://www.top500.org >. Acesso em: 21 set. 2012.

[21] GEIJN, R. A. van de; QUINTANA-ORTÍ, E. S. *The Science of Programming Matrix Computations*. [S.l.]: lulu.com, 2008.

[22] GOTO, K.; GEIJN, R. A. van de. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 2008, ACM, New York, NY, USA, v. 34, n. 3, p. 12:1—-12:25, 2008.

[23] KÅGSTRöM, B.; LING, P.; LOAN, C. van. Gemm-based level 3 blas: high-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Softw.*, 1998, ACM, New York, NY, USA, v. 24, n. 3, p. 268–302, set. 1998.

[24] WHALEY, R. C.; DONGARRA, J. J. Automatically tuned linear algebra software. In: *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. Washington, DC, USA: IEEE Computer Society, 1998. (Supercomputing '98), p. 1–27.

[25] D'ALBERTO, P.; NICOLAU, A. Adaptive winograd's matrix multiplications. *ACM Trans. Math. Softw.*, 2009, ACM, New York, NY, USA, v. 36, n. 1, p. 3:1–3:23, mar. 2009.

[26] RüNGER, G.; SCHWIND, M. Fast recursive matrix multiplication for multi-core architectures. *Procedia Computer Science*, 2010, v. 1, n. 1, p. 67–76, maio 2010.

[27] CHAN, E. et al. Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In: *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 2007. (SPAA '07), p. 116–125.

[28] CHOI, J. et al. Scalapack: a scalable linear algebra library for distributed memory concurrent computers. In: *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*. [S.l.: s.n.], 1992. p. 120 –127.

[29] MPI. *Message Passing Interface*. Disponível em: < `www.mcs.anl.gov/mpi` >. Acesso em: 21 set. 2012.

[30] PVM. *Parallel Virtual Machine*. Disponível em: < `www.csm.ornl.gov/pvm` >. Acesso em: 21 set. 2012.

[31] DONGARRA, J.; LASTOVETSKY, A. L. *High Performance Heterogeneous Computing*. [S.l.]: Wiley, 2009.

[32] LUEBKE, D. et al. Gpgpu: general purpose computation on graphics hardware. In: *ACM SIGGRAPH 2004 Course Notes*. New York, NY, USA: ACM, 2004. (SIGGRAPH '04).

[33] DOERKSEN, M.; SOLOMON, S.; THULASIRAMAN, P. Designing APU Oriented Scientific Computing Applications in OpenCL. In: *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*. [S.l.: s.n.], 2011. p. 587–592.

[34] TAN, G. et al. Fast implementation of DGEMM on Fermi GPU. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA: ACM, 2011. (SC '11), p. 35:1—-35:11.

[35] DU, P. et al. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 2011, n. 0, p. –, 2011.

[36] BUCK, I. et al. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 2004, ACM, New York, NY, USA, v. 23, n. 3, p. 777–786, ago. 2004.

[37] MARK, W. R. et al. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 2003, ACM, New York, NY, USA, v. 22, n. 3, p. 896–907, jul. 2003.

[38] LEE, V. W. et al. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *SIGARCH Comput. Archit. News*, 2010, ACM, New York, NY, USA, v. 38, n. 3, p. 451–460, 2010.

[39] NAKASATO, N. A fast GEMM implementation on the cypress GPU. *SIGMETRICS Perform. Eval. Rev.*, 2011, ACM, New York, NY, USA, v. 38, n. 4, p. 50–55, 2011.

[40] CUI, X.; CHEN, Y.; MEI, H. Improving Performance of Matrix Multiplication and FFT on GPU. In: *Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 2009. (ICPADS '09), p. 42–48.

[41] MATSUMOTO, K. et al. Multi-level optimization of matrix multiplication for gpu-equipped systems. *Procedia Computer Science*, 2011, v. 4, n. 0, p. 342 – 351, 2011.

[42] CUI, X. et al. Auto-tuning Dense Matrix Multiplication for GPGPU with Cache. In: *Proceedings of the 2010 IEEE 16th International Conference on Parallel and Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 2010. (ICPADS '10), p. 237–242.

[43] KURZAK, J.; TOMOV, S.; DONGARRA, J. Autotuning GEMM Kernels for the Fermi GPU. *Parallel and Distributed Systems, IEEE Transactions on*, 2012, PP, n. 99, p. 1, 2012.

[44] SUN, Y.; TONG, Y. CUDA Based Fast Implementation of Very Large Matrix Computation. In: *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2010 International Conference on*. [S.l.: s.n.], 2010. p. 487–491.

[45] QUINTANA-ORTÍ, G. et al. Solving dense linear systems on platforms with multiple hardware accelerators. *SIGPLAN Not.*, 2009, ACM, New York, NY, USA, v. 44, n. 4, p. 121–130, 2009.

[46] ALVAREZ, P. L. et al. Elimination Techniques of Redundant Data Transfers Among GPUs and CPU on Recursive Stream-Based Applications. In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. [S.l.]: IEEE, 2011. p. 708–715.

[47] ALLADA, V.; BENJEGERDES, T.; BODE, B. Performance analysis of memory transfers and GEMM subroutines on NVIDIA Tesla GPU cluster. In: *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*. [S.l.: s.n.], 2009. p. 1–9.

[48] NETPIPE. *Network Protocol Independent Performance Evaluator*. Disponível em: < `www.scl.ameslab.gov/netpipe` >. Acesso em: 21 set. 2012.

[49] FATICA, M. Accelerating linpack with CUDA on heterogenous clusters. In: *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. New York, NY, USA: ACM, 2009. (GPGPU-2), p. 46–51.

[50] MUNSHI, A. *The OpenCL Specification*. 1.2. ed. [S.l.], 2012.

[51] MAGMA. *Matrix Algebra on GPU and Multicore Architectures*. Disponível em: < `icl.cs.utk.edu/magma` >. Acesso em: 21 set. 2012.

[52] BOSILCA, G. et al. Performance Portability of a GPU Enabled Factorization with the DAGuE Framework. In: *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*. [S.l.: s.n.], 2011. p. 395–402.

[53] SILVA, R. I. Á. et al. Minimizing Data Transfers on Matrix Multiplications in GPU-based Heterogeneous Environments. In: *IADIS - International Conference on Applied Computing*, 2012, Madrid.

[54] AMD. *Accelerated Parallel Processing OpenCL Programming Guide*. Maio 2012.

[55] NVIDIA. *CUDA C Best Practices Guide*. Out. 2012.

[56] NVIDIA. *OpenCL Best Practices Guide*. Fev. 2011.

[57] NVIDIA. *Tesla C2075 Computing Processor Board*. Set. 2011.

[58] AMAZON. *Amazon Web Services*. Disponível em: < `http://aws.amazon.com` >. Acesso em: 7 jan. 2013.

ANEXOS

ANEXO A – Artigo Publicado

Trabalho completo sob título "*Minimizing Data Transfers on Matrix Multiplications in GPU-based Heterogeneous Environments*", com autoria de Ricardo I. A. e Silva, Tiago S. Sotana, Rodolfo M. Barros e Jacques D. Brancher, publicado em *IADIS – International Conference on Applied Computing*, 2012, Madrid.

MINIMIZING DATA TRANSFERS ON MATRIX MULTIPLICATIONS IN GPU-BASED HETEROGENEOUS ENVIRONMENTS

Ricardo I. A. e Silva, Tiago Sotana, Rodolfo M. de Barros, Jacques D. Brancher
Computer Department, State University of Londrina (UEL)
Rodovia Celso Garcia Cid, PR 445 km 380, Campus Universitário, Londrina – PR - Brazil

ABSTRACT

Heterogeneous environments in computation systems, composed by a CPU and one or more GPUs, present opportunity in trying to increase performance of linear algebra algorithms. If computational capabilities of each available processor are summed, it can reach respectable few teraFLOPS (TFLOPS). But for it to be possible, a task and its dataset must be parallelized and distributed. Also, data transfers between processors memory must be carefully considered, since they are slow and can hinder global performance. Matrix multiplication is a common task of such systems. It is a highly parallelizable operation and useful, since it is present in most linear algebra algorithms. But its utilization can fall short of performance with processors heterogeneity because of the required high amount of memory transfers. In this work we present a method of distributing tasks and dataset from matrix multiplications that minimizes memory transfers, while trying to utilize 2 GPUs and the CPU. We show that we minimize time spent with transfers from CPU to GPU down to 60% in comparison to execution in single GPU, for very large matrices cases.

KEYWORDS

Matrix Multiplication, Heterogeneous, GPGPU, Parallelism, Memory Transfer

1. INTRODUCTION

Since GPUs became programmable, through the incorporation of shader cores, there have been studies on reusing it for other tasks. It is called General Purpose computing on GPU (GPGPU). Because of its highly parallel nature some scientifically important algorithms presented promising results.

First implementations were done by hacking the existing programming model. Programmers had to map relevant data onto a texture memory object and design the algorithm as a shader program. Because it was unnatural, specific languages and tools for programming GPU were developed. Some of these are Cg, Brook+, CUDA, and OpenCL, with the two latters taking place as the standard.

As of late, increases of performance in CPU with each new generation have been lower than expected. New approaches are being tried, such as increasing number of CPU cores and integrating accelerators, such as FPGA and GPU. Studies show possibilities for utilizing GPGPU as if a specific core of the CPU, concurrently to its own cores. A system with this architecture is regarded as a heterogeneous environment, because it relies on different cores to run the same task.

Matrix multiplication is an important operation of linear algebra, because it is present in most applications of the latter. Also, it is very compute intensive. As such, an optimized algorithm that implements it is the key in achieving performance in others that are its dependents. The most common implementation is delivered by the BLAS library, and had its GEMM interface to be the standard. It is implemented by most other scientifically relevant linear algebra packages, such as MKL, ATLAS, GotoBLAS, FLAME and ACML.

In this work, we present a new approach to distributing tasks extracted from matrix multiplication between available processors in a 2-GPU 1-CPU heterogeneous environment. To check the method we implemented a well-known divide and conquer matrix multiplication.

2. RELATED WORK

Matrix multiplications for CPU architectures are reviewed in Goto and van de Geijn (2008). It explains that matrices should fit into L2 cache and avoid expensive TLB misses. Works Rünger and Schwind (2010) and Quintana-Ortí et al. (2009) addresses implementations on threaded and multi-core architectures. Because there are differences between processors and each new processor generation brings needs to optimize again, paper D'Alberto and Nicolau (2009) and ATLAS library take on automatically tuned routines. They are all relevant in extracting most FLOPS from CPU, which are still core element in heterogeneous environments.

The other important element, GPGPU, is profiled for linear algebra in Volkov and Demmel (2008). Cui et al. (2009), Sun and Tong (2010), Tan et al. (2011) and Nakasato (2011) all work on coding optimizations for matrix multiplication kernels for various GPUs, from which computation reached TFLOPS performance ceiling in a single processor. Cui et al. (2010), Bosilca et al. (2011) and Du et al. (2011), the latter through MAGMA library, worked on portability of kernel performance between distinct GPU architectures, both from different generations and vendors. Lastly, Allada et al. (2009) showed relevancy of memory transfers and trustworthiness of OpenCL profiling tools on that subject.

On the actual field of matrix multiplication on heterogeneous environments with GPGPU, Ohshima et al. (2007) developed a method of parallelize tasks based on performance of each processor, so that ideally both spend the same time computing. It works well on some cases, but it does not minimize memory transfers nor the amount of data to be transferred. Igual et al. (2011) developed a run-time scheduler capable of decomposing linear algebra operations, parallelize tasks, executing out-of-order based on dependencies and is GPGPU aware. It delegates resolution of matrix multiplication to specialized BLAS libraries.

3. PARALLELIZATION ON HETEROGENEOUS ENVIRONMENTS

Heterogeneous environments are characterized by having distinct processors working on the same system. Common occurrences of this architecture are systems built around a general purpose CPU, added by discrete GPUs, through a high speed interconnection bus. In such cases, each processor has its own memory space, thus it is a distributed memory system.

Also, it is possible in this kind of system to execute algorithms that involve every processor, be it CPU or GPU. This means that algorithms can be parallel in processor level. To achieve that, data has to be modeled so it can be explicitly divided between memory spaces. More importantly, programmers have to carefully consider memory transfers when designing algorithms.

On current systems, memory transfers between CPU and GPUs usually are the bottleneck in many algorithms implementations. GPUs already reached few tera of float-point operations per second (TFLOPS) in theoretical peak of computational power, and memory bandwidth of ~300GB/s in its own memory space. At the same time, the best CPU–GPU interconnection implementation, the PCIe 3.0, has theoretical bandwidth peak of only 16GB/s.

Table 1. Specification of the system used in experiments

CPU	Core i5 760 2.8GHz
Memory	4GB DDR3 1333MHz
OS	Windows 7 (64 bits)
GPU	Radeon HD 6850
Graphics Bus	PCIe 2.0
VRAM	1GB
GPU processor clock	820MHz
GPU memory clock	1050MHz

For this reason, to better take advantage of GPGPU potential in heterogeneous environments, memory transfers must be minimized. Figure 1 shows the performance of the system described in table 1, when executing the SGEMM routine from *clAmdBlas* library on a single GPU and utilizing *clEnqueue*[*Read*|*Write*]*Buffer()* calls from OpenCL to handle memory transfers. It works on two single-precision matrices, each sizing 1000x1000 where elements have 4 bytes, which configures a dataset of 8MB (1000x1000x4x2). Also the subroutine achieves ~750 GFLOPS and the system has a CPU-to-GPU optimal

bandwidth of 5.15GB/s and a GPU-to-CPU of 5.87GB/s. The test shows both bandwidths barely reach half of that optimal. The dataset starts on CPU memory space, is transferred to GPU memory space, then gets back to CPU in form of a single results 1000x1000 matrix, sizing 2MB.

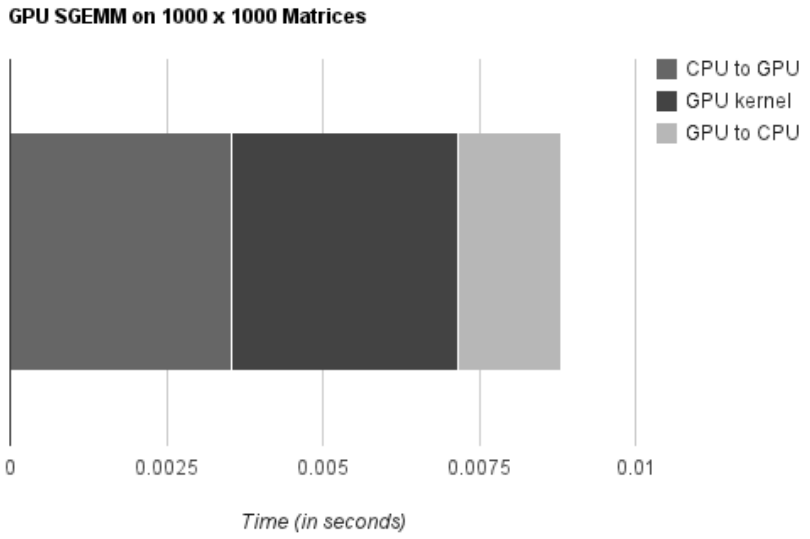


Figure 1: Performance profile of a GPGPU routine.

4. MATRIX MULTIPLICATION

The most common and simple algorithm, also known as naïve matrix multiplication, is the traditional three nested loops. It has asymptotic computational cost of $O(n^3)$ and, also, its memory access patterns are not best suited to the way most processors fetch memory. Added to that, it is easy to have big enough matrices so that they all don't fit into highest level memory, within memory hierarchies. All these characteristics contribute to make the naïve implementation slower.

There is a class of algorithms that involve blocking. Matrices are divided into blocks of smaller submatrices, which gets multiplied and then reassembled. By dividing and operating with smaller matrices, it is possible to solve most or all of the problems present in naïve implementation. That way, they are able to achieve better performance, even considering extra overheads that are introduced.

There are blocking algorithms such as Strassen and Coppersmith-Winograd. Their asymptotic computational costs are, respectively, $O(n^{2.802})$ and $O(n^{2.3727})$. Even so, Winograd is rarely used because of a considerable overhead hidden in the Big–O notation. Although smaller, for the same reason Strassen is only used when dealing with large matrix multiplications. Their overhead comes in terms of extra float-point operations (flops, from now on), memory space utilization and, consequently, memory operations (memops).

There is the possibility of blocking matrices by using a divide-and-conquer variation of the naïve algorithm. It will still has asymptotic cost of $O(n^3)$. And, like in Strassen and Winograd, it introduces overhead in terms of flops and memops, but smaller. And it still gives the benefits of blocking already discussed. Also, as we show, by employing it there is the possibility of reducing the amount of data to be transferred to each processor involved in the execution of the algorithm.

4.1 Blocking Scheme

In this section, we present the divide-and-conquer algorithm utilized. Consider a matrix multiplication given by $C = AB$, where $A \in R^{m \times k}$, $B \in R^{k \times n}$ and $C = R^{m \times n}$, with $\{m, n, k\} \in N^*$. Matrices are divided so that

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \times \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} C_1 & C_2 \\ C_3 & C_4 \end{bmatrix},$$

with each $A_i \in R^{\frac{m}{2} \times \frac{k}{2}}$, $B_i \in R^{\frac{k}{2} \times \frac{n}{2}}$ and $C_i \in R^{\frac{m}{2} \times \frac{n}{2}}$. From there, we proceed with naïve matrix multiplication to extract the following set of equations

$$\begin{aligned} C_1 &= A_1 B_1 + A_2 B_3, \\ C_2 &= A_1 B_2 + A_2 B_4, \\ C_3 &= A_3 B_1 + A_4 B_3, \\ C_4 &= A_3 B_2 + A_4 B_4. \end{aligned}$$

Then, we choose to represent C_i elements of matrix C in terms smaller matrix multiplications between A_i and B_i, so that

$$\begin{aligned} C_1 &= M_1 + M_2, \\ C_2 &= M_3 + M_4, \\ C_3 &= M_5 + M_6, \\ C_4 &= M_7 + M_8. \end{aligned}$$

From that, we are able to define a set $M = \{M_1, \dots, M_8\}$. These are the smaller matrix multiplications needed to resolve the original. They are given by

$$\begin{aligned} M_1 = A_1 B_1,\ M_2 = A_2 B_3,\ M_3 = A_1 B_2,\ M_4 = A_2 B_4, \\ M_5 = A_3 B_1,\ M_6 = A_4 B_3,\ M_7 = A_3 B_2,\ M_8 = A_4 B_4. \end{aligned}$$

So, we segment operations involved in $C = AB$ in two classes: multiplications and additions.

4.2 Execution

To avoid unnecessary memory transfers, we maximize memory reuse. So, if a matrix block is sent to a device, it should run all tasks that the specific matrix block is involved. And, because this would be impossible on a regular matrix multiplication, we choose to ignore matrix additions.

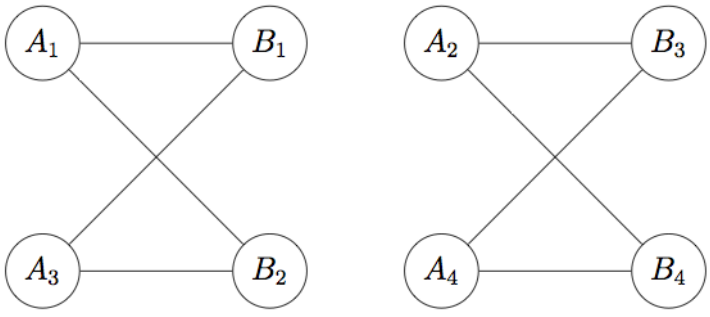


Figure 2: Relationship graph between matrices blocks that are factors of a same multiplication.

With this in mind, we model a graph G = (V, E) representing the relationship between matrices blocks, where vertices set V are matrices blocks and edges set E represents participation as factors in a multiplication. Such graph is depicted in figure 2. From the picture, it is clear that there are two sets of independent blocks.

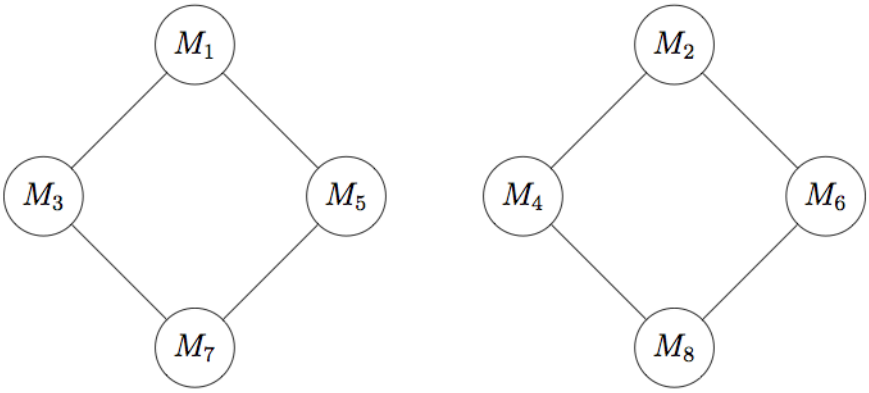


Figure 3: Graph depicting multiplication operations that depend on a same factor.

Because multiplications only have two factors, edges E represent not only the relationship of factor blocks, but a multiplication. So we can clarify even further by defining multiplications graph $G_m = (V_m, E_m)$, where vertices $V_m = E$ and edges $E_m = V$, depicted in figure 3. It shows which multiplications should be executed on the same processor.

With this procedure, there are three groups of tasks, as shown by figure 4. Two are independent set of multiplications and one is a set of matrix additions that depends on results of the multiplications. What we gather from it is that multiplications should be executed by accelerator devices. When finished, they return results to CPU that execute inexpensive additions and, thus, reassembling the resulting matrix.

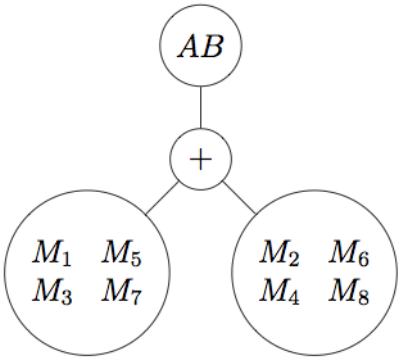


Figure 4: Tree representing extracted tasks in naïve blocking matrix multiplication.

5. EXPERIMENTS

For experimentation to the proposed method, we implemented an algorithm, called 'hybrid', which receives already parted matrices, distribute blocks to processors accordingly, send operation commands, receive results blocks and merge them in a single matrix by adding on CPU. For that, our implementation assumes usage of hierarchical matrices, which are already perfectly parted.

Our implementation utilized clAmdBlas 1.8 SGEMM routine, with AMD APP SDK 2.7, when executing multiplication on GPU. To comply with the SDK, communications with GPU were done through OpenCL calls. We also implemented a naïve CPU threaded matrix addition, through OpenMP framework, for use in merging results coming from GPU.

Since we did not readily have two equal GPUs, we assembled our system with two distinct, the other being an Radeon HD 4830. While it is not capable of running our hybrid method, because it is not supported by clAmdBlas, we tested the interference of both GPU's on each other at data transfers between CPU and GPU. We found that interference was fewer than 3%, since both together could not reach PCIe 2.0 bandwidth

peak. Also, because even two Radeon HD 6850 would not reach this peak, we assumed our system to have two of that same GPU. So we configured our method to execute only a group of multiplications.

Because we aimed at reducing data transfer time from CPU to GPU, we compare our method to a straight call to clAmdBlas' SGEMM. For that end, we utilized available profiling tools from OpenCL run-time platform. With them we profiled timing of stages of execution, similar to that of figure 1. Experiments ran on system described in table 1.

Table 2. Experiments timings.

Method	Size	CPU to GPU (s)	GPU kernel (s)	GPU to CPU (s)	CPU addition (s)
Hybrid	500	0.0020	0.0011	0.0023	0.0003
	1000	0.0030	0.0020	0.0028	0.0016
	2000	0.0079	0.0144	0.0071	0.0070
	4000	0.0224	0.0798	0.0203	0.0277
GPU only	500	0.0015	0.0006	0.0008	
	1000	0.0038	0.0036	0.0017	
	2000	0.0113	0.0199	0.0048	
	4000	0.0371	0.1544	0.0172	

5.1 Results

Timings gathered from experimentation are shown in table 2. Method column indicate method utilized in the following row. Size column indicate matrices sizes, the same for all dimensions($m = n = k$). CPU to GPU column indicates time to transfer every input data from CPU memory space to GPU, in seconds. GPU kernel column indicate time for the GPU to finish calculations, in seconds. GPU to CPU column indicates time to transfer results back from GPU memory space. Lastly, CPU addition column indicates time spent in merging results by adding matrices received from GPU, in seconds. Obviously, since this stage is only present in hybrid method, only its rows have results for that column. Data is also displayed on figure 5, for better visualization.

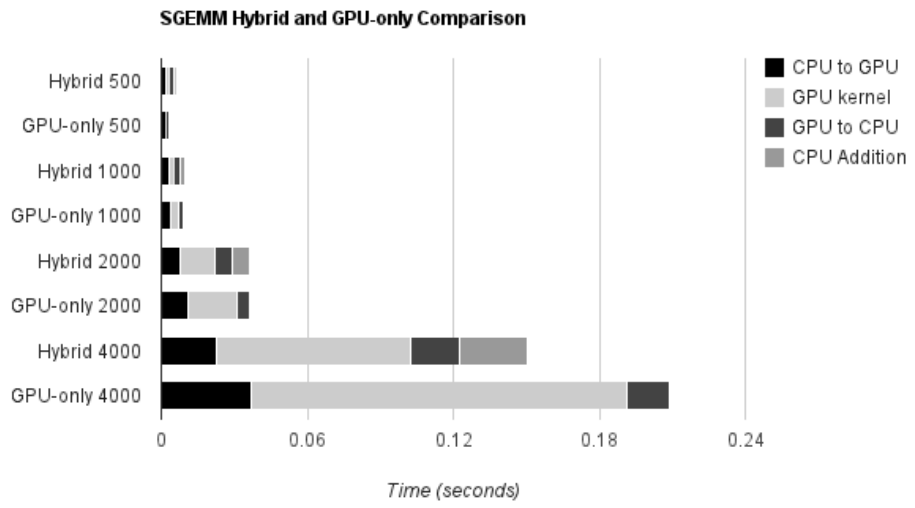


Figure 5: Timings from experiments side-by-side for comparison.

5.2 Discussion

From figure 5 we see that currently our method only truly benefits when multiplying large matrices. But we also see that matrices addition on CPU do take a considerable amount of time. We are confident that there are possible optimizations in this area, both in code and by employing a better processor.

We confirmed the efficiency of our methodology about memory transfers from CPU to GPU for cases of matrices sizes greater than 1000×1000. At that size, the said transfer for hybrid method is about 79% of the time of GPU-only. At 4000 sizes, hybrid is about 60% of GPU-only. Also, we note the presence of overhead in both kinds of transfers in hybrid method compared to GPU-only. This is because the latter only issue one copy command on a single block of dataset on memory, while the former must issue four copy commands.

6. CONCLUSION

In this work, we present a new approach to distributing tasks extracted from matrix multiplication between available processors in a 2-GPU 1-CPU heterogeneous environment so that each GPU receives the minimum necessary dataset for operating in parallel with the other. Through the results of our implementation, we prove the efficiency of that method for large matrices. Starting from 1000×1000 matrix multiplications, time for transfers from CPU to GPU are reduced by 21%. And it can reduce down to 60%, in 4000×4000 matrices size case.

Now we set to investigate if that gain still holds true in cases where usage of two GPUs saturates available PCIe bandwidth. Also, we should investigate if rectangular-shaped memory transfers from CPU to GPU do introduce significant overhead, as is available in OpenCL calls. Since it still is unusual for matrices on memory to be perfectly hierarchy-parted, this would be a wanted feature, if viable.

REFERENCES

Allada, V. et al. 2009. Performance analysis of memory transfers and GEMM subroutines on NVIDIA Tesla GPU cluster. *In Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–9.

Bosilca, G. et al, 2011. Performance Portability of a GPU Enabled Factorization with the DAGuE Framework. *In Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 395–402.

Cui, X. et al, 2009. Improving Performance of Matrix Multiplication and FFT on GPU. *In Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems, ICPADS '09*, pages 42–48, Washington, DC, USA.

Cui, X. et al 2010. Auto-tuning Dense Matrix Multiplication for GPGPU with Cache. *In Proceedings of the 2010 IEEE 16th International Conference on Parallel and Distributed Systems, ICPADS '10*, pages 237–242, Washington, DC, USA.

D'Alberto, P. and Nicolau, A., 2009. Adaptive Winograd's matrix multiplications. *ACM Trans. Math. Softw.*

Du, P. et al, 2011. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*.

Goto, K. and van de Geijn, R. A., 2008. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*

Igual, F. D. et al, 2011. The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations. *Journal of Parallel and Distributed Computing*, 2011.

Nakasato, N., 2011. A fast GEMM implementation on the cypress GPU. *SIGMETRICS Perform. Eval. Rev.*

Ohshima, S. et al, 2007. Parallel processing of matrix multiplication in a CPU and GPU heterogeneous environment. *In Proceedings of the 7th international conference on High performance computing for computational science, VECPAR'06*, pages 305–318, Berlin, Heidelberg.

Quintana-Ortí, G. et al, 2009. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw.*

Rünger, G. and Schwind, M., 2010. Fast recursive matrix multiplication for multicore architectures. *Procedia Computer Science*.

Sun, Y. and Tong, Y., 2010. CUDA Based Fast Implementation of Very Large Matrix Computation. *In Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2010 International Conference on*, pages 487–491.

Tan, G. et al, 2011. Fast implementation of DGEMM on Fermi GPU. *In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11*, pages 35:1-35:11, New York, NY, USA.

Volkov, V. and Demmel, J. W., 2008. Benchmarking GPUs to tune dense linear algebra. *In Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC '08*, pages.31:1-31:11, Piscataway, NJ, USA.