UNIVERSIDADE FEDERAL FLUMINENSE

BERNARDO BREDER

Ordenação da Submissão de Kernels Concorrentes para Maximizar a Utilização dos Recursos da GPU

NITERÓI

2016

UNIVERSIDADE FEDERAL FLUMINENSE

BERNARDO BREDER

Ordenação da Submissão de Kernels Concorrentes para Maximizar a Utilização dos Recursos da GPU

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Redes de Computadores e Sistemas Distribuídos e Paralelos

Orientador:
LÚCIA M. A. DRUMMOND

Co-orientador:
ESTEBAN CLUA

NITERÓI

2016

BERNARDO BREDER

Ordenação da Submissão de Kernels Concorrentes para Maximizar a Utilização dos Recursos da GPU

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Redes de Computadores e Sistemas Distribuídos e Paralelos

Aprovada em Setembro de 2016.

BANCA EXAMINADORA

Prof. Lúcia M. A. Drummond - Orientador, UFF

Prof. Esteban Walter Gonzalez Clua - Co-orientador, UFF

Prof. Cristiana Barbosa Bentes, UERJ

Prof. Maria Cristina Silva Boeres, UFF

Niterói

2016

Agradecimentos

Agradeço primeiramente a Deus por ter nos dado a força e a perseverança necessárias para nos mantermos firmes na direção do nosso objetivo.

Agradeço aos meus pais Giovanni Gargano Breder e Vanda Neves Breder, ao meu irmão Raphael Breder e a minha esposa Paula Teixeira Tavares Monteiro Breder que sempre acreditaram e me apoiaram em todos os meus momentos e decisões.

Agradeço aos meus familiares e amigos que, direta ou indiretamente nos ajudaram a transformar esse sonho em realizada.

Agradeço aos meus amigos de classe e professores que fizeram parte da minha vida durante esses anos.

Agradeço aos meus orientadores Lúcia Drummond, Esteban Clua e da ajuda do grupo de estudo em GPU Cristiana Bentes, Eduardo Charles, Rommel Anatoli e Gabriel Gazolla.

Resumo

Arquiteturas mais recentes de GPUs permitem que kernels sejam executados de forma concorrente no mesmo hardware. Este recurso incrementa bastante o poder destes dispositivos, uma vez que viabiliza a concorrência de execuções de tarefas diferentes no mesmo hardware. Entretanto, as decisões de hardware sobre a ordem de execução depende fortemente da ordem com que os kernels são submetidos, fazendo com que o aproveitamento do hardware não seja otimizado. Este trabalho propõem uma nova abordagem de otimização para ordenar os kernels submetidos focando na maximização dos recursos utilizados, melhorando a média do *turnaround time*. O modelo de simulação dos kernels com os recursos do hardware será modelado como um Problema da Mochila onde é usado o algoritmo Guloso e Programação Dinâmica. Os resultados são medidos usando kernels com diferentes tamanhos e recursos requeridos e demonstram um ganho significativo na média do *turnaround time* e *system throughput* comparado a uma execução padrão de kernels concorrentes.

Palavras-chave: gpu, ordenação, kernel, concorrente, maximizar, utilização, recursos, escalonador, otimização

Abstract

New generation of Architectures GPUs allow concurrent execution kernels on the same hardware. This feature can increase the power of these devices, since it makes the concurrent execution of different tasks on the same hardware possible. However, the hardware decisions on the execution order depend heavily on the order at which the kernels are submitted, causing the utilization of the hardware to be not optimized. This work proposes a novel optimization approach to reorder the kernels submittion focusing on maximizing the resources utilization, improving the average turnaround time. The model the simulation of the kernels with the hardware resources represented as a knapsack problem and an algorithm greedy and dynamic programming approach to solve them. The results are measured using kernels with different sizes and resource requirements, resulting on significant gains in the turnaround time average and system throughput compared with a standard execution of concurrent kernels.

Keywords: gpu, order, kernel, concurrent, maximize, utilization, resources, scheduler, otimization

Lista de Figuras

1.1 Arquitetura CPU e GPU [7] 2

1.2 Concorrência de kernels na arquitetura Fermi [28] 2

1.3 Execução concorrente entre kernels de mesma cor na arquitetura Fermi [6] onde o eixo x representa o tempo 3

1.4 Execução concorrente entre kernels de mesma cor na arquitetura Kepler [6] onde o eixo x representa o tempo 3

1.5 Diferentes ordens de submissão dos kernels nas 2 filas. A organização na parte (a) mostra uma taxa de ocupação menor porque no instante que o kernel k_3 está executando, o kernel k_4 não consegue executar por não ter recursos disponível. Já a organização na parte (b), a taxa de ocupação é melhor e o tempo final de execução também. 4

2.1 A GPU possui uma quantidade maior de circuitos de ALU (*Arithmetic Logical Units*), representado pela caixa verde, perante a CPU. Essa quantidade indica a característica da GPU processar mais dados paralelamente em operações aritiméticas. Em contrapartida, a GPU possui um circuito muito limitado de unidade de controle e memória, representado pela caixa amarela e laranja. Isso representa uma limitação do circuito de controle e memória perante a CPU. 7

2.2 Arquitetura de uma Kepler GK110 com 15 SMX 8

2.3 Arquitetura de uma SMX da Kepler GK110 com 192 CUDA *cores* 9

2.4 A concorrência da Fermi [31] é limitada a uma única fila de trabalho. Na Kepler, as *streams* podem executar concorrentemente usando filas separadas de tarefas 10

2.5 Diferenças de concorrência entre a Fermi e a Kepler [30]. A linha do tempo é representado pela seta verde e as filas de trabalho são representados pelas caixas em cinza chamado de Stream 1, 2 e 3. No exemplo, foi criado 3 filas de tarefa, onde a primeira fila está o trabalho representado pelas letras A, B e C, a segunda fila representado pelos trabalhos P, Q, R e a terceira pelos trabalhos X, Y e Z. Essas filas mostram a ordem de execução das tarefas e todas as filas podem ser executados concorrentemente. 10

2.6 Arquitetura da Maxwell com 6 grupos de GPC [32] 11

2.7 Hierarquia de memória da GPU 12

2.8 Particionamento de um kernel para executar em qualquer arquitetura [26] . 13

4.1 $N = 4$ Kernels com diferente instantes $T_\beta \mid \beta \in \{0, 1, 2, 3, 4\}$ na simulação do calculo da ordem de execução. 21

4.2 Taxa de ocupação $TO_{\alpha\beta}$ dos exemplos de ordenação Ord_α para cada instante T_β e também apresenta uma linha tracejada com a média ponderada da $TO_{\alpha\beta}$ 22

4.3 Instante T_i dos dois examples de ordenação 23

4.4 Instante T_N e a construção da lista utilizando o modo *Round-Robin* para cada ordenação vista anteriormente 23

5.1 *Trace* de execução da ordenação *Standard* com 64 kernels 35

5.2 *Trace* de execução da ordenação *Greedy* com 64 kernels 35

5.3 *Trace* de execução da ordenação *Dynamic* com 64 kernels 35

5.4 Resultado do ANTT para $N = 32$ e $N = 64$ na TitanX. 36

5.5 Resultado do ANTT para $N = 128$ e $N = 256$ na TitanX. 36

5.6 Resultado do STP para $N = 32$ e $N = 64$ na TitanX. 37

5.7 Resultado do STP para $N = 128$ e $N = 256$ na TitanX. 37

5.8 Resultado do ANTT na K40 para $N = 32$ e $N = 64$ 39

5.9 Resultado do ANTT na K40 para $N = 128$ e $N = 256$ 39

5.10 Resultado do STP na K40 para $N = 32$ e $N = 64$ 40

5.11 Resultado do STP na K40 para $N = 128$ e $N = 256$ 40

Lista de Tabelas

1.1 Exemplo de kernels com seus respectivos recursos e tempo de duração . . . 4

3.1 Trabalhos relacionados de escalonamento da GPU 18

4.1 Valores de x_i para cada instante T_i 21

4.2 Taxa de ocupação para cada instante T_i 21

4.3 Variáveis utilizadas nos algoritmos 26

4.4 Valores de w_i e v_i para cada kernel k_i 30

4.5 Calculo do valor de $OPT(k_4, 58)$ 30

4.6 Calculo do valor de $OPT(k_5, 59)$ 30

5.1 Configuração das GPUs nos resultados 32

5.2 *Overhead* O_{greedy} do Guloso na TitanX 38

5.3 *Overhead* $O_{dynamic}$ da Programaçã Dinâmica na TitanX 38

Sumário

1 Introdução **1**

1.1 Contextualização 1

1.2 Escalonamento das tarefas 3

2 Arquitetura da GPU **6**

2.1 Arquitetura 6

2.2 Filas de execução 7

2.3 Hierarquia de Memória 11

2.4 Portabilidade de execução 13

3 Trabalhos Relacionados **14**

3.1 União de kernels 14

3.2 Divisão de kernels 15

3.3 Contextos de aplicação 16

3.4 Ordenação de submissões 16

3.5 Sintetizando os trabalhos 17

4 Ordenação das Submissões de Kernels **19**

4.1 Definição do Problema 19

4.2 Processo de ordenação 20

4.3 Problema da Mochila 0-1 Multidimensional (PMM) 23

4.4 Adaptação do problema da mochila para o problema de reordenação de Kernels 24

4.5 Simplificação do Problema 25

4.6 Algoritmo para ordenação de kernels 25

4.6.1 Algoritmo Guloso 27

4.6.2 Programação Dinâmica 28

4.6.3 Exemplo da construção de Matriz 29

4.6.4 Vantagens e Desvantagens do Guloso e da Programação Dinâmica . 31

5 Resultados Experimentais 32

5.1 Ambiente Computacional 32

5.2 Metodologia 33

5.3 Desempenho 34

5.3.1 Average Normalized Turnaround Time 35

5.3.2 System Throughput 36

5.4 Custo de execução do método proposto 37

5.5 Portabilidade do método entre diferentes arquiteturas 38

5.5.1 Average Normalized Turnaround Time 39

5.5.2 System Throughput 39

6 Conclusões 41

Referências 43

Capítulo 1

Introdução

1.1 Contextualização

As *Graphics Processing Units* (GPUs) foram inicialmente desenvolvidas para realizar o trabalho de renderização de informações bidimensionais e tridimensionais para exibição de objetos gráficos para o usuário. O avanço de sua capacidade de processamento tornou possível que pudessem passar a processar outros tipos de estrutura de dados, além de vértices e pixels, tornando-as em excelente alternativa para realizar computação paralela com um custo baixo.

Owens et al. [33] apresentou o termo *General-Purpose Computation* para discutir aplicações não gráficas que podem usufruir do poder computacional das GPUs para resolver problemas de desempenho de algoritmos. Naquele momento, programar para GPU significava expressar os algoritmos da aplicação em operações sobre estruturas de dados gráficos (especialmente *pixels* e vetores). O processo de conversão de dados da aplicação para este domínio, quando possível, representava um esforço grande de processamento que podia onerar a solução. Porém, quando ocorria um ganho de desempenho da execução do algoritmo, esse esforço era compensado.

Com esse quadro adaptativo das aplicações e com o surgimento de vários trabalhos sendo desenvolvidos utilizando essas GPUs, os fabricantes começaram a alterar a arquitetura da GPUs para introduzir um conjunto de instruções de propósito geral e foram criadas ferramentas, *frameworks* e linguagem de programação (CUDA, OpenCL) para usufruir melhor desse potencial computacional.

A NVIDIA desenvolveu uma arquitetura chamada *Compute Unified Device Architecture* (CUDA), capaz de trabalhar para possibilitar o uso em conjunto da CPU e GPU no

processamento de aplicações de propósito geral, possibilitando assim a utilização da GPU tanto para renderização de gráficos quanto para outros propósitos de fins genéricos.

O desenvolvimento de aplicações com a plataforma CUDA utiliza da CPU para controlar o fluxo de execução sequencial dos algoritmos e utiliza da GPU para efetuar grandes quantidades de operações aritméticas em matrizes ou vetores de forma paralela [25][9][7]. A figura 1.1 mostra o trabalho colaborativo entre CPU e GPU interligados por um barramento.

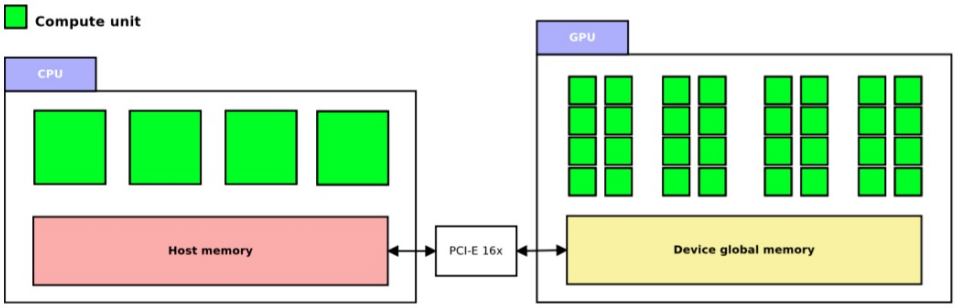

Figura 1.1: Arquitetura CPU e GPU [7]

Com o objetivo de maximizar o desempenho das aplicações, o uso da GPU vem crescendo cada vez mais. Dessa forma, as aplicações estão compartilhando de forma mais abrangente os recursos da GPUs para atender às suas necessidades. A arquitetura Fermi da NVIDIA foi a pioneira na funcionalidade de execução de kernels concorrentes na GPU. Porém, apenas submissões de kernels no mesmo contexto de aplicação poderiam executar concorrentemente [45][15][28]. A figura 1.2 ilustra uma execução sequencial e concorrente.

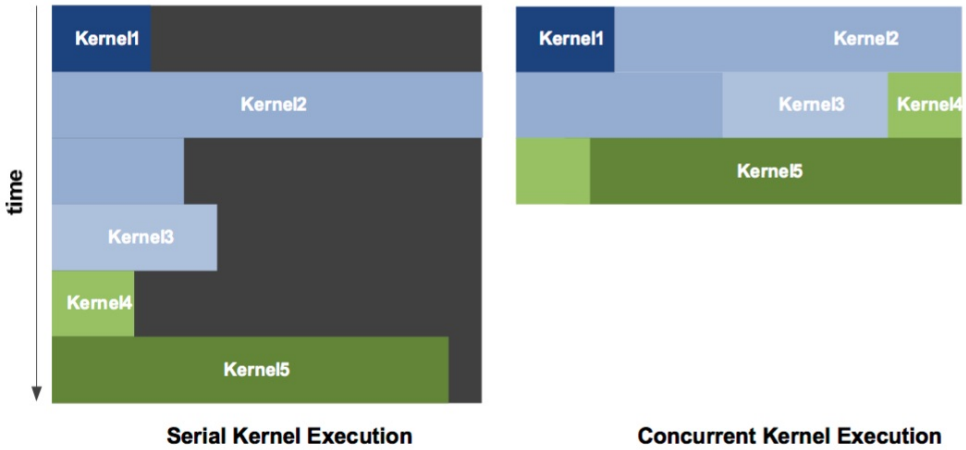

Figura 1.2: Concorrência de kernels na arquitetura Fermi [28]

Em função da inclusão de concorrências de kernels na arquitetura Fermi limitada por uma única fila de trabalho e um único contexto de aplicação, surgiram diversos trabalhos

para simular a execução concorrente em diferentes contextos de aplicação [15][45]. Em seguida, o lançamento da tecnologia Hyper-Q, já na arquitetura Kepler [30], possibilitou que diferentes contextos de aplicações executassem algoritmos concorrentemente. A figura 1.3 mostra a limitação de uma única fila de trabalho da Fermi [6] onde um conjunto de kernels de mesma cor podem e devem executar concorrentemente, mas não são executados juntos por conta da única fila de trabalho. A figura 1.4 mostra a execução na Kepler [6] com o mesmo conjunto de kernels, onde todos da mesma cor são executados concorrentemente.

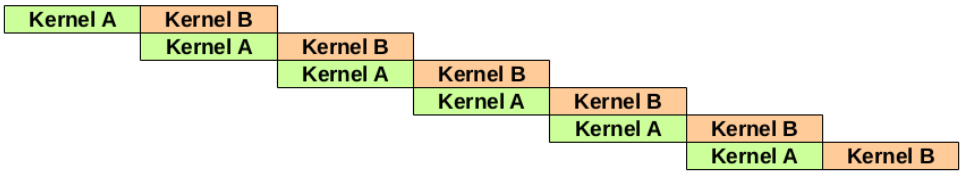

Figura 1.3: Execução concorrente entre kernels de mesma cor na arquitetura Fermi [6] onde o eixo x representa o tempo

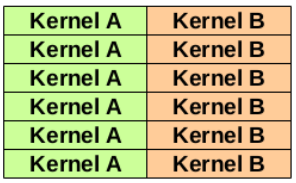

Figura 1.4: Execução concorrente entre kernels de mesma cor na arquitetura Kepler [6] onde o eixo x representa o tempo

Com a inclusão da tecnologia Hyper-Q na Kepler, os kernels enviadas para a GPU passaram a ser executados concorrentemente mas sem haver um bom escalonamento das tarefas na GPU. A ordem com que as tarefas são submetidas é a ordem com que elas são executadas. Essa característica parece ser adequada quando se deseja ter um maior controle da ordem com que os kernels são executados, mas não é vantajoso quando se deseja maximizar a utilização da GPU.

1.2 Escalonamento das tarefas

A política de escalonamento dos kernels na GPU é proprietária da NVIDIA e nenhuma informação está documentada. Existem, entretanto, especulações que a política de escalonamento segue a estratégia *left-over*[34][49][37]. O escalonador dispara os blocos do kernel na primeira fila de *hardware*, se houver recursos suficientes. Os próximos disparos são feitos nas próximas filas de *hardware* seguindo o estilo *round-robin*, quando há recursos

disponíveis. Dois kernels da mesma fila de *hardware* são executados na ordem sequencial. O problema dessa política de escalonamento é que a ordem que os kernels são inseridos nas filas de *hardware* determina a taxa de ocupação da GPU.

A Tabela 1.1 ilustra um exemplo de 4 kernels com seus respectivos percentuais de utilização de recursos e tempo de execução. Essa porcentagem unidimensional representa uma simplificação do problema para ilustrar o exemplo. Em um ambiente com 2 filas, ilustrado pela figura 1.5, ao submeter os kernels $\{k_2, k_1, k_3, k_4\}$ nesta ordem, o kernel k_2 será submetido na fila 1 e o k_1 será submetido na fila 2. Quando o k_2 terminar, o k_3 será submetido para a fila 2 por possuir recursos disponível neste momento. Porém, quando o kernel k_1 terminar, o kernel k_4 não terá recurso disponível ocupado pelo kernel k_3. Quando o kernel k_3 terminar, o kernel k_4 inicializará sua execução. Porém se considerarmos uma ordem de submissão diferente $\{k_4, k_1, k_3, k_2\}$, a taxa de ocupação será melhor e o tempo final de execução também.

Exemplo Didático	Kernels			
	k_1	k_2	k_3	k_4
Recursos (w_i)	30%	30%	50%	60%
Tempo (t_i)	$20ms$	$15ms$	$15ms$	$15ms$

Tabela 1.1: Exemplo de kernels com seus respectivos recursos e tempo de duração

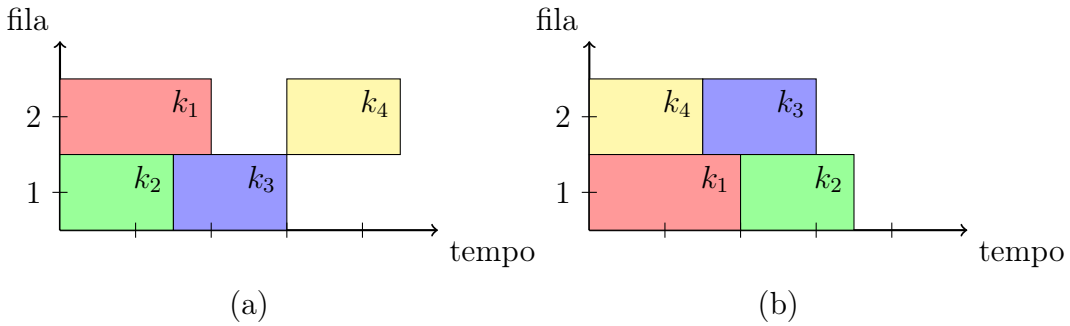

Figura 1.5: Diferentes ordens de submissão dos kernels nas 2 filas. A organização na parte (a) mostra uma taxa de ocupação menor porque no instante que o kernel k_3 está executando, o kernel k_4 não consegue executar por não ter recursos disponível. Já a organização na parte (b), a taxa de ocupação é melhor e o tempo final de execução também.

O trabalho irá propor uma estratégia de ordenação para submissão de múltiplos kernels, com o objetivo de melhorar o *turnaround time* médio dos kernels. A proposta da ordenação é escolher um conjunto de kernels que combinados utilizam melhor os recursos da GPU de forma concorrente. Para isso, utilizamos o problema da mochila [2][38][23][3][4] para efetuar a seleção, sendo que o resultado representa a maximização do uso dos recursos dos kernels. A capacidade da mochila foi associado à quantidade de recursos da GPU

e os itens da mochila foram mapeados aos kernels. Como cada kernel possui uma quantidade de recursos necessários, o peso dos itens da mochila foram modelados como sendo os recursos necessário de cada kernel e o valores de cada item da mochila foi mapeado como a média das porcentagem dos recursos por unidade de tempo. A quantidade de memória compartilhada, número de registradores e número de threads representam os recursos do kernel.

O restante do trabalho está organizado da seguinte forma. O Capítulo 2 introduz a arquitetura da GPU. O Capítulo 3 apresenta os trabalhos relacionados com execução concorrente de kernels na GPU. O Capítulo 4 apresenta a proposta do trabalho. O Capítulo 5 apresenta os resultados de diversos testes realizados. Por fim, o Capítulo 6 apresenta a conclusão do trabalho com discussões sobre os resultados.

Capítulo 2

Arquitetura da GPU

2.1 Arquitetura

As GPUs são co-processadores dedicados para processamento gráfico da classe SIMD (*Single Instruction Multiple Data*). Elas são muito adequadas para algoritmos que requerem um intenso volume de cálculo matemático de inteiro e ponto flutuante, especialmente em algoritmos que podem ser trivialmente paralelizados por dados.

Para facilitar a *interface* entre o programador e a GPU, a linguagem CUDA fornece abstrações simples sobre a hierarquia de threads, memória e sincronização. O modelo de programação em CUDA permite ao desenvolvedor escrever programas denominados de kernels sem a necessidade de conhecer previamente detalhes do tipo de *hardware* onde será executado. A arquitetura CUDA suporta diversas linguagens de programação, incluindo C, Fortran e OpenCL. A linguagem CUDA tem sido amplamente utilizado por aplicações e trabalhos de pesquisa encontrados em diferentes dispositivos [48].

Uma diferença essencial entre CPU e GPU, é que a primeira dedica a maior quantidade do seu circuito na unidade de controle e de cache, mostrado na figura 2.1 a segunda dedica mais o circuito na ALU (*Arithmetic Logical Units*), o que a torna muito mais eficaz para cálculos matemáticos [11][48]. A figura 2.1 ilustra a diferença do circuito da CPU e da GPU.

Uma GPU possui vários SMs (*Stream Multiprocessor*), que por sua vez executam grupos de *Threads*, chamados de *Warps*. Cada SM possui vários núcleos chamados de *CUDA cores*. Cada um dos *CUDA cores* possui um *pipeline* de operações aritméticas ALU e de ponto flutuante FPU (*Floating Point Unit*).

Uma GPU Kepler GK110 [30], por exemplo, possui 15 SMs [10]. Cada geração de

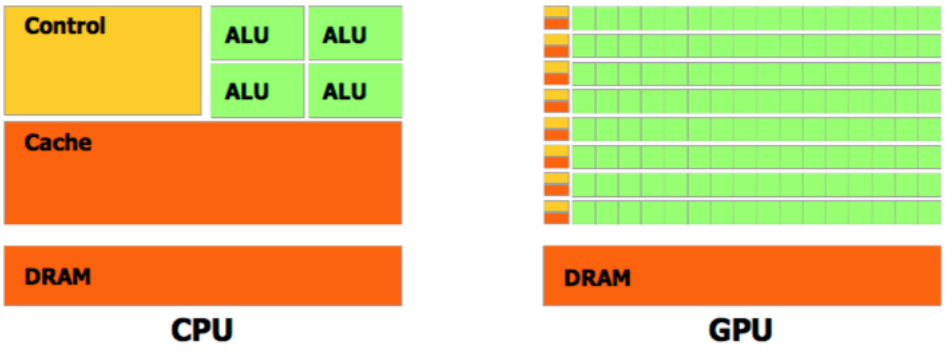

Figura 2.1: A GPU possui uma quantidade maior de circuitos de ALU (*Arithmetic Logical Units*), representado pela caixa verde, perante a CPU. Essa quantidade indica a característica da GPU processar mais dados paralelamente em operações aritiméticas. Em contrapartida, a GPU possui um circuito muito limitado de unidade de controle e memória, representado pela caixa amarela e laranja. Isso representa uma limitação do circuito de controle e memória perante a CPU.

GPUs implementa variações na arquitetura dos seus SMs, sendo comum dar-lhes nomes diferentes. Na arquitetura Kepler, por exemplo, o SM é denominado de SMX. Todos os SMs possuem acesso a uma memória global que pode ser compartilhada por todos os outros SMs e também pode ser escrita pela CPU quando se deseja transferir dados de entrada para a execução de uma kernel. A figura 2.2 mostra a arquitetura da Kepler GK110.

Cada SM da Kepler GK110 possui 192 *CUDA cores*, com 48K de memória própria chamada Memória Compartilhada ou *L1 Cache* e uma memória somente leitura chamada *Read-Only Data Cache*. A memória compartilhada possui um tempo de acesso muito menor do que a memória global. A memória somente de leitura é usada para memória de textura onde os dados são para leitura [30]. A figura 2.3 mostra a arquitetura de uma SMX da Kepler GK110.

2.2 Filas de execução

A configuração de *hardware* de uma GPU que especifica o número de cores por SM, volume de memória e funcionalidades, depende da arquitetura utilizada em sua fabricação e pode variar dependendo do modelo. A capacidade de computação, ou *compute capability*, é um parâmetro que especifica as principais características de hardware que estarão disponíveis para o desenvolvedor. Cada capacidade de computação é expressada na forma de número aonde a arquitetura Tesla está entre 1.0 a 1.3, Fermi entre 2.0 a 2.1, Kepler entre 3.0 e 3.7, Maxwell entre 5.0 a 5.3.

Figura 2.2: Arquitetura de uma Kepler GK110 com 15 SMX

A cada evolução da capacidade de computação, novas funcionalidades são criadas. A arquitetura Fermi possui uma única fila de trabalho, onde todos os kernels submetidos são executados de forma sequencial. A tecnologia Hyper-Q introduzida na arquitetura Kepler 32 filas de trabalho, possibilitando assim que até 32 kernels possam ser executados concorrentemente.

As filas de execução concorrentes são representadas pelas Streams. Cada Stream representa um fila que executa uma sequencia de kernels na ordem FIFO (*First In First Out*). Assim, o kernel k_{i+1} só poderá ser executado quando o kernel k_i terminar. Entretanto, diferentes Streams podem executar concorrentemente caso hajam recursos disponíveis [34]. Na arquitetura Kepler, a GPU possui somente 32 filas de *hardware*, não tendo limite bem definido de filas de Streams. Dessa forma, quando são criados mais Streams do que a quantidade de filas de *hardware*, a GPU compartilha mais de uma Stream numa mesma fila de *hardware* [34][30].

Dependendo da GPU, um conjunto de kernels em diferentes Streams podem ser executados de forma diferente. A figura 2.5 mostra um modelo de execução de uma mesma submissão para GPU em diferentes arquiteturas. Na Fermi, apenas as tarefas (C,P) e

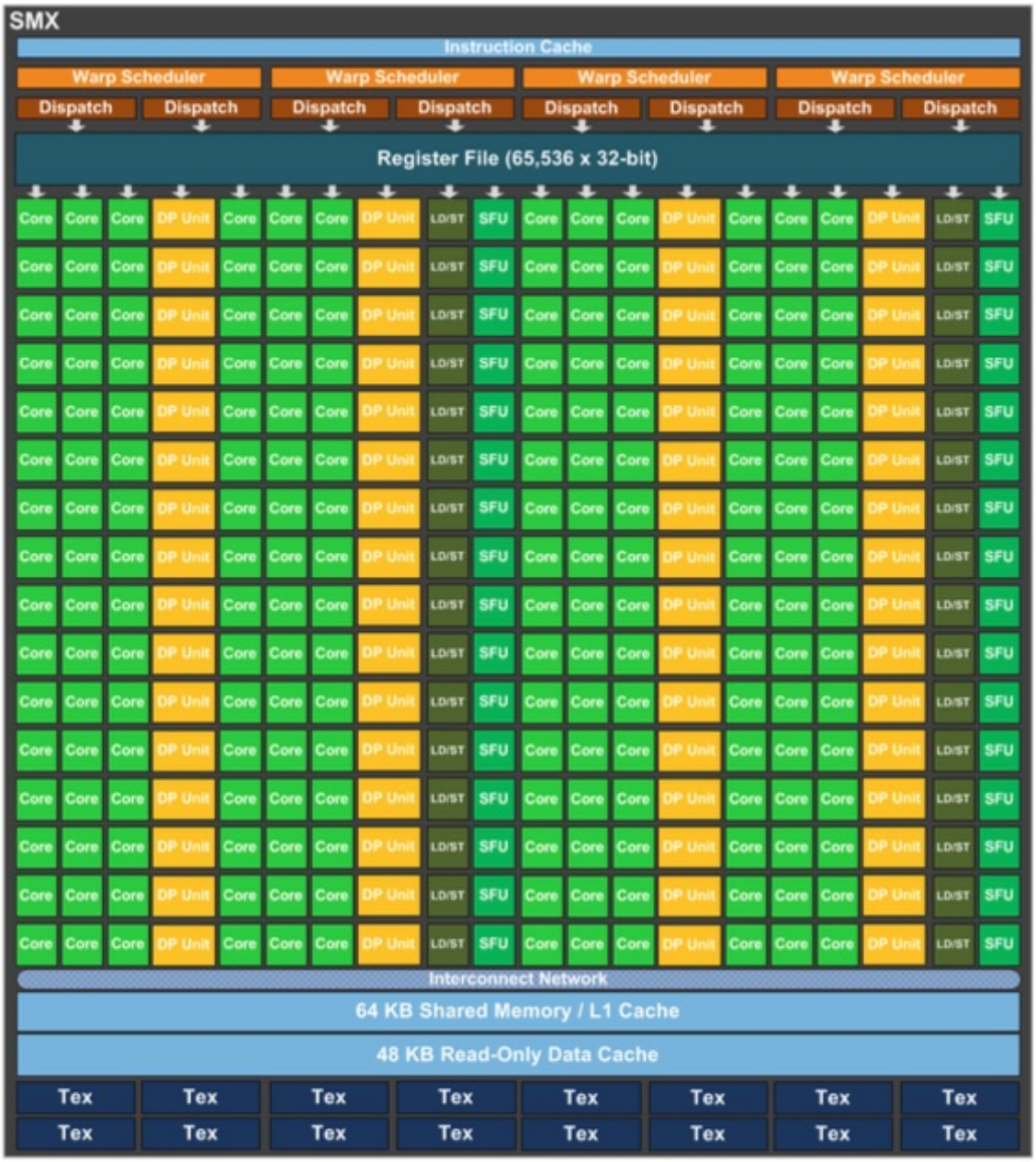

Figura 2.3: Arquitetura de uma SMX da Kepler GK110 com 192 CUDA *cores*

(R,X) podem executar concorrentemente em função da dependência de *streams* causada pela única fila de trabalho. No modelo da Kepler, as três *streams* executam concorrentemente [30].

A tecnologia Hyper-Q oferece um benefício significativo no uso de sistemas computacionais paralelos baseado em MPI (*Message Passing Interface*). A utilização de MPI para fazer acesso as GPUs pode trazer compartilhamento do recurso. No caso da arquitetura Fermi, esse compartilhamento é prejudicado pela função de dependência de *streams* causada pela única fila de trabalho. A tecnologia Hyper-Q possibilitou retirar essa falsa

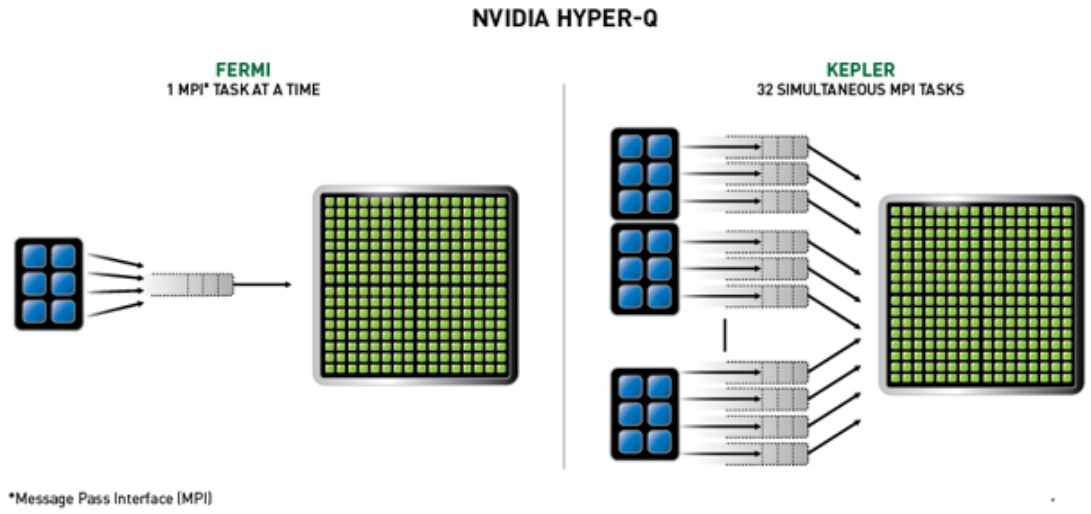

Figura 2.4: A concorrência da Fermi [31] é limitada a uma única fila de trabalho. Na Kepler, as *streams* podem executar concorrentemente usando filas separadas de tarefas

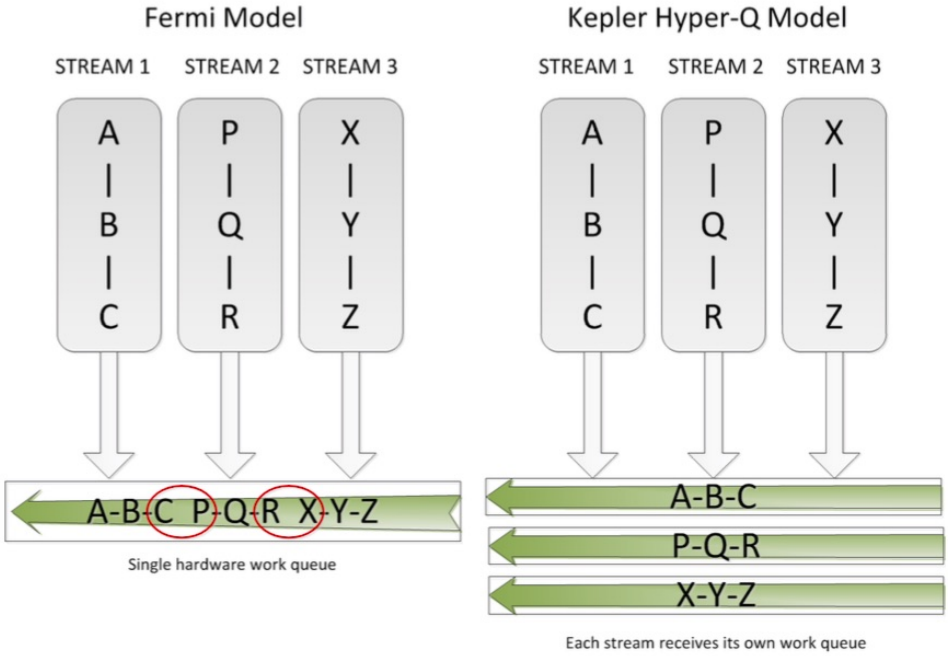

Figura 2.5: Diferenças de concorrência entre a Fermi e a Kepler [30]. A linha do tempo é representado pela seta verde e as filas de trabalho são representados pelas caixas em cinza chamado de Stream 1, 2 e 3. No exemplo, foi criado 3 filas de tarefa, onde a primeira fila está o trabalho representado pelas letras A, B e C, a segunda fila representado pelos trabalhos P, Q, R e a terceira pelos trabalhos X, Y e Z. Essas filas mostram a ordem de execução das tarefas e todas as filas podem ser executados concorrentemente.

dependência aumentando dramaticamente a eficiência do compartilhamento da GPU nos processos do MPI [19].

Após a arquitetura Kepler, a Maxwell [32] foi lançada com um aumento de 2 vezes o

desempenho por *watts* [10]. Além disso, 6 grupos de SMs foram criados chamados de GPC (*Graphic Processors Clusters*) na placas mais atuais como TitanX. Cada SM é chamado de SMM e contém 128 *CUDA cores* com 96K de memória compartilhada. A figura 2.6 ilustra a arquitetura com os 6 grupos de SMs.

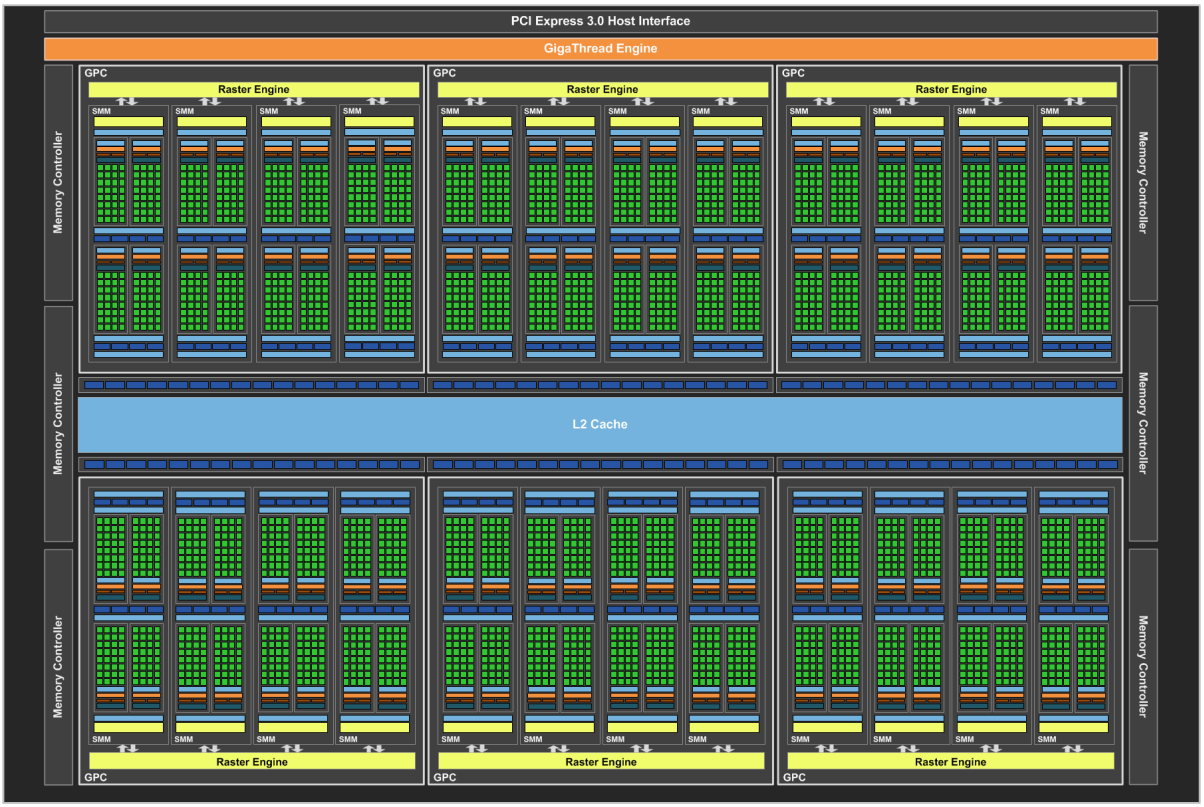

Figura 2.6: Arquitetura da Maxwell com 6 grupos de GPC [32]

2.3 Hierarquia de Memória

A GPU possui espaço de memória própria onde é organizado hierarquicamente, de tal forma que quando convenientemente explorado, pode ser um elemento importante para o bom desempenho da aplicação.

A figura 2.7 mostra as localizações das diferentes memórias tanto da CPU quanto da GPU. A imagem ilustra um contexto de execução de 1 grade, com 2 blocos (0,0) e (0,1) com 2 threads cada um, dando um totla de 4 threads. Na parte de baixo da imagem pode ser observado a memória global e a memória constante onde podem ser acessado a partir da CPU por meio da API CUDA usada pelas aplicações para efetuar troca de dados.

A memória constante pode ser acessado pela GPU como somente leitura com baixa capacidade de armazenamento e alta velocidade de acesso perante a memória global. Por

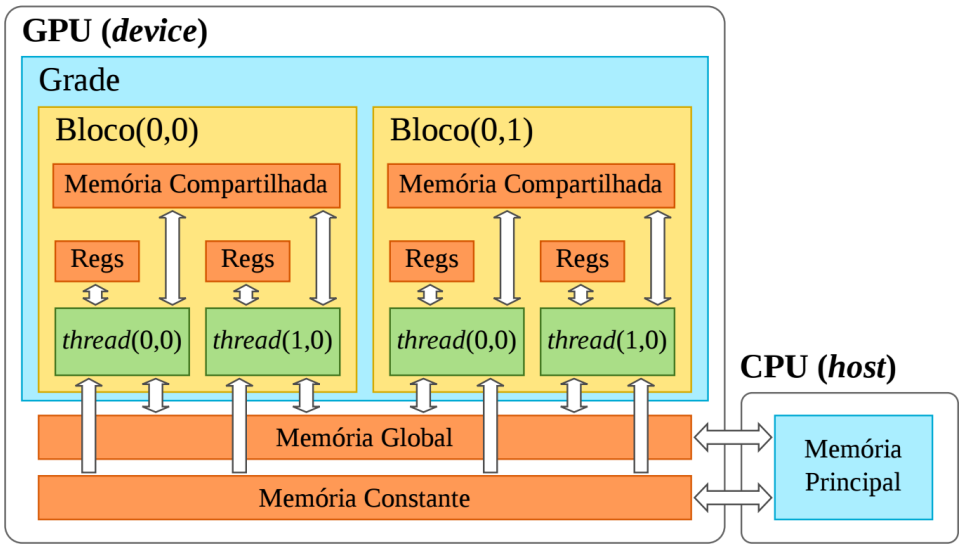

Figura 2.7: Hierarquia de memória da GPU

sua vez, a memória global pode ser lido e escrito pela GPU e apresenta uma capacidade de armazenamento similar a memória principal da CPU. Os dados armazenados na memória global podem ser acessados por todas as threads e ficam disponíveis durante o tempo de execução da aplicação. Apesar da ampla capacidade de armazenamento, a memória global apresenta alta latência de acesso representando um componente crítico a ser considerado no projeto de aplicação baseadas em GPU.

Cada multiprocessador possui registradores, uma quantidade de memória compartilhada com baixa latência de acesso e uma memória de cache (cache L1). Os dados armazenados em registradores são de acesso restrito as threads individuais, sendo usado para armazenar dados de variáveis locais por exemplo. A memória compartilhada pode ser acessada por todas as threads da mesma SM e possui uma alta eficiencia de acesso e compartilhamento de dados.

As características de cada tipo de memória da GPU representam uma importância para a aplicação e para o desenvolvedor. A memória global é ampla, porém lenta; enquanto que a memória compartilhada é escassa, porém extremamente rápida. Com isso, com o uso bem adequado da memória representa um aspecto marcante para o desempenho do kernel.

2.4 Portabilidade de execução

O modelo de programação em CUDA permite desenvolver kernels que possam ser executados em diferentes arquiteturas e configurações de GPUs. A linguagem CUDA possibilita a transparência da quantidade de *CUDA cores* e de recursos da GPU. Esse fator é importante para possibilitar a portabilidade dos kernels para as novas arquiteturas e para a escalabilidade da execução.

Para que essa transparência ocorra, o modelo de programação em CUDA orienta ao programador particionar o programa em subprogramas menores que podem ser resolvidos em blocos de execução paralelo. Essa decomposição possibilita que os subprogramas possam executar em qualquer SM que estiver disponível, em qualquer ordem, simultaneamente ou sequencialmente de modo que um programa CUDA pode ser executado em qualquer número de SMs, como ilustrado na figura 2.8. A figura 2.8 mostra um kernel sendo particionado em bloco para serem executados independentemente.

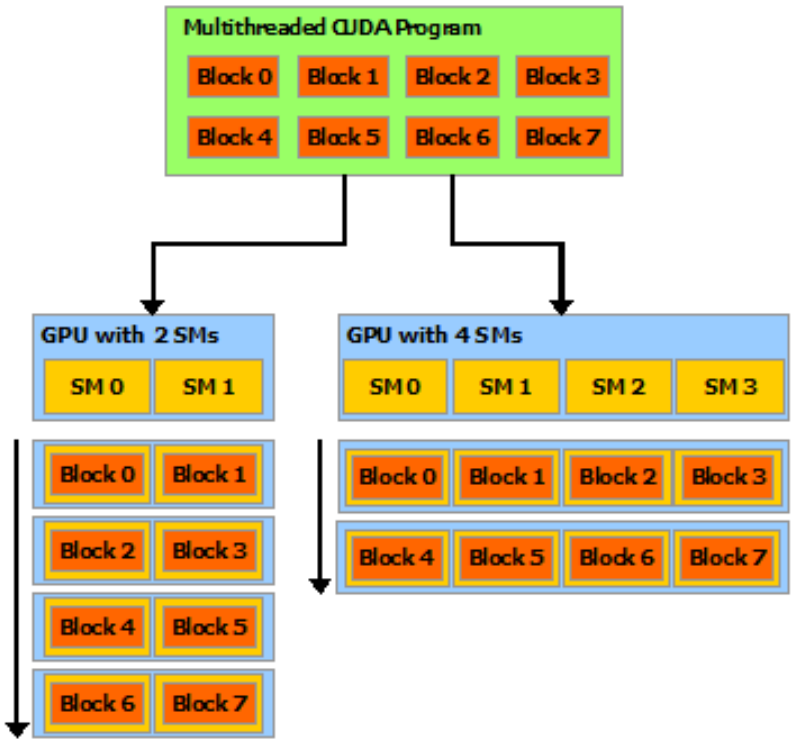

Figura 2.8: Particionamento de um kernel para executar em qualquer arquitetura [26]

Capítulo 3

Trabalhos Relacionados

O crescente uso de GPUs e o aumento da necessidade de acesso compartilhado vem demandando cada vez mais a necessidade de gerenciar o acesso concorrente aos recursos da GPU. Alguns trabalhos apresentam soluções para otimizar ou melhorar a execução de diferentes kernels, compartilhando os recursos da GPU. Esses trabalhos apresentam características semelhantes e podem ser categorizado como: União de kernels, Divisão de kernels, Contextos de aplicação e Ordenação de kernels.

3.1 União de kernels

Nas primeiras tentativas de simulação de kernels concorrentes, alguns trabalhos propõem a fusão de kernels de aplicações distintas em um único kernel. Essa estratégia não propõe nenhuma mudança de *hardware* possibilitando com que seja implantada em arquiteturas mais antigas.

O trabalho do Peters et al. [36] consiste em executar um kernel, chamado de persistente, onde cada bloco de threads observa em um laço infinito um espaço de memória dedicado esperando por uma leitura de argumentos de função para ser executados. Neste trabalho, o kernel persistente fica esperando tarefas para ser executado. Quando a encontra, realiza a tarefa e transfere a resposta para o solicitador. Através desse mecanismo, o trabalho permite executar kernels concorrentemente em blocos separados.

Outros esforços foram feitas para simular concorrência a nível de *software* realizando *merging* de código de kernels. O trabalho do Guevara et al. [16] propõe um sistema de compilação que une dois kernels em um. Neste trabalho apenas kernels que apresentarem possibilidade de compartilhar recursos serão unidos.

O trabalho de G. Wang et al. [44] une códigos de kernels com o objetivo de reduzir o consumo de energia, através de uma heurística. Esse trabalho propõe um método para computar o nível de consumo de energia de uma kernel. Os detalhes do consumo de energia são detalhados em Zhu et al. [50].

O trabalho do Gregg et al. [15] propõe um escalonamento de kernels concorrentes em OpenCL, utilizando uma abordagem diferente de fila de trabalho para escalonar as tarefas. O escalonador implementa uma heurística de fila para alocar tarefas e aumentar os benefícios de concorrência. Esse trabalho constrói dois tipos de fila de trabalho: A primeira utiliza uma abordagem de *round-robin* para associar a tarefa à fila, onde cada kernel entra seguindo o critério FCFS (*First-come, first-served*). Já o critério da segunda fila assume que o percentual de recursos de cada kernel é fixo se eles estiverem na mesma fila.

Este grupo de soluções caracteriza-se por adotar arquiteturas que ainda não suportam concorrência de kernels.

3.2 Divisão de kernels

Em uma outra direção, alguns autores propõem mecanismos de modificar a granularidade dos kernels com o objetivo de melhorar a taxa de utilização da GPU. Zhong et al. [49] propõem um sistema chamado *kernelet* que em tempo de execução divide os kernels, em problemas menores e escalona-os de tal forma a melhorar a taxa de ocupação da GPU com a execução de kernels concorrentes. Para isso, o *kernelet* utiliza do algoritmo Guloso para escalonar os pequenos pedaços dos kernels.

Pai et al. [34] propõe uma melhoria desta abordagem através de uma técnica de *elastic kernel*. Para isso, o trabalho realiza uma transformação do código dos kernels para que eles possam utilizar melhor as características de *hardware* da GPU.

Existem também esforços para dividir os recursos da GPU entre os kernels concorrentes chamado *spatial*. Liang et al. [21] usa o CUDA *profiler* para determinar a quantidade de memória global solicitada para cada kernel e calcula o comportamento do kernel em termos de largura de banda latência e de memória, executando cada kernel com uma kernel de ajuste. Sua proposta foca não apenas no particionamento dos SMs com as Thread, mas também em decidir quais kernels irão executar concorrentemente. Os autores usam uma abordagem heurística baseada em programação dinâmica para dividir os SMs e selecionar os kernels para a execução simultânea.

O trabalho realizado pelo Adriaens et al. [1], também utiliza do termo *spatial* para dividir os recursos utilizados pelos kernels de tal forma a melhorar a execução concorrente utilizando modelo de escalonamento preemptivo.

Tanasic et al. [40] propõe um conjunto de extensões do hardware para permitir a execução preemptiva dos kernels de tal forma a melhorar o uso dos recursos da GPU em função da política de escalonamento implementado pelo trabalho.

O trabalho feito pelo Jiao et al. [18] propõe a combinação de execução concorrente de kernels com a técnica de gerencia de energia chamado *Dynamic Voltage and Frequency Scaling* (DVFS) para melhorar o consumo de energia da execução dos kernels.

As propostas levantadas nesta categoria propõem mudanças de hardware para melhorar a concorrência entre os kernels, mas este trabalho utiliza de mecanismos de software para melhorar o acesso concorrente dos kernels na GPU.

3.3 Contextos de aplicação

Nas arquiteturas mais recentes (Fermi, Kepler e Maxwell) o processamento concorrente passou a ser suportado pelo *hardware*, embora na Fermi a concorrência apenas seja possível se os kernels estiverem no mesmo contexto de aplicação.

O trabalho do L. Wang et al. [45] explora a execução de kernels de diferentes contextos de aplicação através de um contexto compartilhado chamado *context funneling*. Através dessa abordagem, esse contexto compartilhado é usado como serviço para executar kernels de diferentes contextos de aplicação.

O Ravi et al. [37] apresenta um *framework* para aplicações de contexto diferentes executarem numa máquina virtual de forma transparente, com o objetivo de compartilhar uma ou mais GPUs. Além disso, esse trabalho propõe uma forma de combinar kernels para melhor um aumento no desempenho. Neste trabalho propõem-se uma estratégia para calcular esta afinidade baseando nas suas características que não conflitam.

3.4 Ordenação de submissões

Finalmente, os trabalho que propõem alcançar o máximo de utilização com base na ordem em que kernels são invocados para GPU, chamado de reordenamento de kernel, são os que mais se aproximam com a atual proposta. Wende et al. [47] propõe uma pre-ordenação

dos kernels que irão ser submetidos para o hardware Fermi, onde a execução concorrente de kernel usam apenas uma fila de tarefas. Os autores criam um escalonador que organiza a fila das tarefas utilizando *round-robin*. Embora o trabalho tenha alcançado bons resultados com o aumento da concorrência intercalando diferentes filas de execução, o hardware das GPUs modernas têm o mecanismo Hyper-Q, que implementa essas filas em hardware.

Li et al. [20] propõem um reordenação para a novas arquiteturas de GPU com tecnologia Hyper-Q. Sua abordagem tenta executar os kernels com a utilização de recursos complementares utilizando o conceito de rodadas de execução. O trabalho procura cumprir um rodada com kernels usando um algoritmo guloso para o problema da multidimensional *bin-packing*. O uso de rodadas de execução simplifica as decisões de programação. Embora este seja o trabalho que mais se aproxime da atual proposta, em nossa técnica procuramos buscar por duas soluções de ordenação evitando a sincronização implícita da conclusão das rodadas, resolvendo o problema a cada vez quando há recursos disponíveis.

3.5 Sintetizando os trabalhos

Todos os trabalhos realizados desde 2009 a 2015 vem trazendo propostas de melhoria de hardware e soluções de software com o objetivo de melhorar o acesso aos recursos da GPU e sua utilização. Cada trabalho na tabela 3.1 apresenta diferentes linhas de solução em diferentes arquiteturas, onde a primeira coluna mostra o autor e o ano do trabalho, a segunda coluna mostra a categoria do trabalho enumerada no inicio desse capítulo, a terceira coluna mostra a arquitetura que foi testada e por fim, a quarta coluna mostra os *benchmark* e simuladores utilizados nos trabalhos para mostrar os resultados.

Trabalho	Categoria	Arquitetura	Teste
Guevara et al., 2009 [16]	União	Tesla	10 Parboil benchmark
Wang et al., 2010 [44]	União	Tesla	CUDA SDK SPEC2K benchmark
Peters et al., 2010 [36]	União	Tesla	MatMul e Sorting
Gregg et al., 2012 [15]	União	Fermi	Bitonic Sort, Kmeans Floyd Warshall Nearest Neighbor Simple Convolution Fast Walsh Transform
Zhong et al., 2014 [49]	Divisão	Fermi	CUDA SDK Parboil Benchmark CUSP
Pai et al., 2013 [34]	Divisão	Fermi	Parboil2 benchmark
Liang et al., 2015 [21]	Divisão	Kepler	CUDA SDK
Adriaens et al., 2012 [1]	Divisão	Tesla	Microbenchmarks Parboil benchmark
Jiao et al., 2015 [18]	Divisão	Kepler	CUDA SDK Rodinia Benchmark
Wang et al., 2011 [45]	Contexto	Fermi	Scalable Synthetic Compact Application
Ravi et al., 2011 [37]	Contexto	Fermi	Image Processing PDE Solver, K-Means Binomial, BlackScholes K-Nearest Neighbours Molecular Dynamics
Tanasic et al., 2014 [40]	Contexto	Kepler	Parboil Benchmark
Park et al., 2015 [35]	Contexto	Kepler	Rodinia Benchmark Parboil Benchmark
Wende et al., 2012 [47]	Ordenação	Fermi	Molecular Dynamics Simulation
Li et al., 2015 [20]	Ordenação	Kepler	BlackScholes SmithWaterman Electrostatics Embarrassingly Parallel

Tabela 3.1: Trabalhos relacionados de escalonamento da GPU

Capítulo 4

Ordenação das Submissões de Kernels

4.1 Definição do Problema

Diferentes kernels podem ser executados em qualquer ordem concorrentemente desde que a GPU tenha recursos suficientes para executar. Quando um kernel não conseguir executar concorrentemente, por não haver recursos disponíveis, ficará aguardando numa fila de execução até que algum recurso seja liberado.

A ordem com que os kernels são submetidos indica a ordem da alocação de recursos. Dependendo da combinação de recursos alocados pelos kernels, determinado pela ordem de execução, a GPU pode ficar com baixa taxa de ocupação e com alto tempo de *turnaround time* médio. A figura 1.5 e a Tabela 1.1 mostram duas diferentes ordens de execução dos kernels gerando diferentes taxa de ocupação da GPU.

Dada uma GPU D e dada a operação de ordenação dos kernels $Ord = \{k_1, ..., k_N\}$, os kernels serão submetidos na ordem calculada Ord para a GPU D. Considere que D tenha um número de SMs igual a NSM e que os recursos de cada SM sejam representados por $R_j \mid j \in \{1, 2, 3\}$, onde:

- R_1 representa o tamanho da memória compartilhada de cada SM
- R_2 representa o número de registradores de cada SM
- R_3 indica o número máximo de número de threads de cada SM

Uma variável W_j é utilizada para representar a capacidade da GPU D que é o somatório das capacidades de cada SM. Portanto, a equação 4.1 mostra o valor da capacidade total do dispositivo D.

$$W_j = (R_j + 1) \times NSM \mid j \in \{1, 2, 3\} \tag{4.1}$$

Para cada kernel k_i, os recursos necessários para serem usados em sua execução será definido por $w_{ij} \mid i \in \{1, ..., N\}, j \in \{1, 2, 3\}$, o número de blocos do kernel b_i e o tempo estimado t_i^{est} do kernel será definido como:

- t_i^{est}: tempo estimado de execução
- b_i: o número de blocos
- r_{i1}: a quantidade de memória compartilhada por bloco
- r_{i2}: o número de registradores por bloco
- r_{i3}: o número de threads por bloco
- $w_{i1} = r_{i1} \times b_i$: a quantidade de memória compartilhada de todos os blocos
- $w_{i2} = r_{i2} \times b_i$: o número de registradores de todos os blocos
- $w_{i3} = r_{i3} \times b_i$: o número de threads de todos os blocos

A quantidade de memória compartilhada, número de registradores, número de threads e o número de blocos podem ser obtidos através de ferramenta de *Profiler* da NVIDIA[29]. O tempo estimado da execução de um kernel pode ser obtido de uma execução anterior ou por técnicas de estimativa de tempo[22].

4.2 Processo de ordenação

Para que o conjunto de N kernels submetidos para a GPU tenha um bom *turnaround time* médio, os kernels devem ser organizados de tal maneira que a cada instante $T_\beta \mid \beta \in \{0, ..., N\}$, o *turnaround time* seja bom ou ótimo. Mesmo tentando otimizar o *turnaround time* para todo T_β, essa estratégia não garante o *turnaround time* ótimo no contexto global. Portanto, o problema será tratado de tal forma que a cada instante T_β um conjunto de kernels serão selecionados com um bom ou ótimo *turnaround time* médio.

A figura 4.1 ilustra os $N = 4$ kernels da Tabela 1.1 com dois exemplos de ordenações $Ord_\alpha \mid \alpha \in \{a, b\}$ e com seus instantes T_β de cada kernel simulado. As ordenações Ord_α dos kernels podem ser representados pela Tabela 4.1, onde para cada instante T_β,

existe valores para $x_i \mid i \in \{1, ..., N\}$, onde $x_i = 1$ significa que o k_i foi escalonado para executar no instante T_β. A Tabela 4.2 mostra as taxas de ocupação $TO_{\alpha\beta} \mid \alpha \in \{a, b\}, \beta \in \{0, ..., N\}$ descrita na equação 4.2 das ordenações Ord_α para cada instante T_β e o cálculo da média ponderada WAT (*Weighted Average Time*) ilustrada na equação 4.2 descrito na equação 4.3. A tabela 4.2 também mostra que a Ord_b tem um WAT maior e que o tempo de execução total foi reduzido.

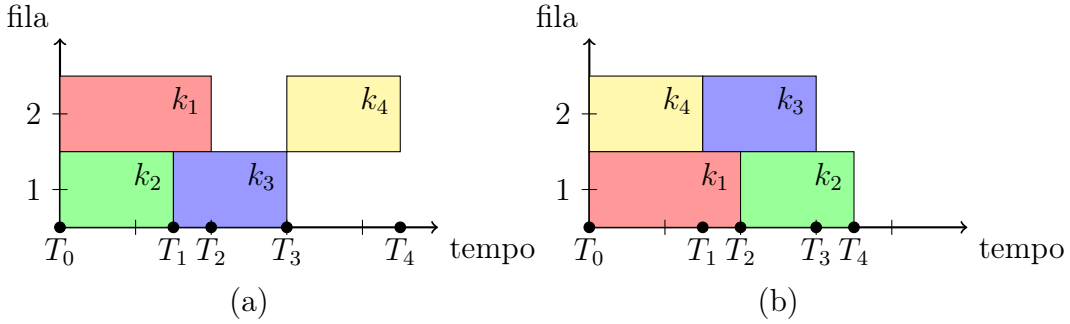

Figura 4.1: $N = 4$ Kernels com diferente instantes $T_\beta \mid \beta \in \{0, 1, 2, 3, 4\}$ na simulação do calculo da ordem de execução.

	Ord_a				Ord_b			
$x_{\alpha i}$	x_{a1}	x_{a2}	x_{a3}	x_{a4}	x_{b1}	x_{b2}	x_{b3}	x_{b4}
T_0	1	1	0	0	1	0	0	1
T_1	1	0	1	0	1	0	1	0
T_2	0	0	1	0	0	1	1	0
T_3	0	0	0	1	0	1	0	0
T_4	0	0	0	0	0	0	0	0

Tabela 4.1: Valores de x_i para cada instante T_i

$$TO_{\alpha\beta} = \sum_{i=1}^{N} w_i x_{\alpha i} \times (T_{\beta+1} - T_\beta) \tag{4.2}$$

$$WAT_\alpha = \sum_{\beta=0}^{N} TO_{\alpha\beta} \div \sum_{i=1}^{N} t_i^{est} \tag{4.3}$$

$TO_{\alpha\beta}$	T_0	T_1	T_2	T_3	T_4	WAT_α
Ord_a	$60\% \times 15ms$	$80\% \times 5ms$	$50\% \times 10ms$	$60\% \times 15ms$	$0\% \times 0ms$	60%
Ord_b	$90\% \times 15ms$	$80\% \times 5ms$	$80\% \times 10ms$	$30\% \times 5ms$	$0\% \times 0ms$	77%

Tabela 4.2: Taxa de ocupação para cada instante T_i

O processo de ordenação de kernels consiste na simulação da execução do escalonamento de todos eles. Para que a simulação possa ser construída é preciso primeiro criar

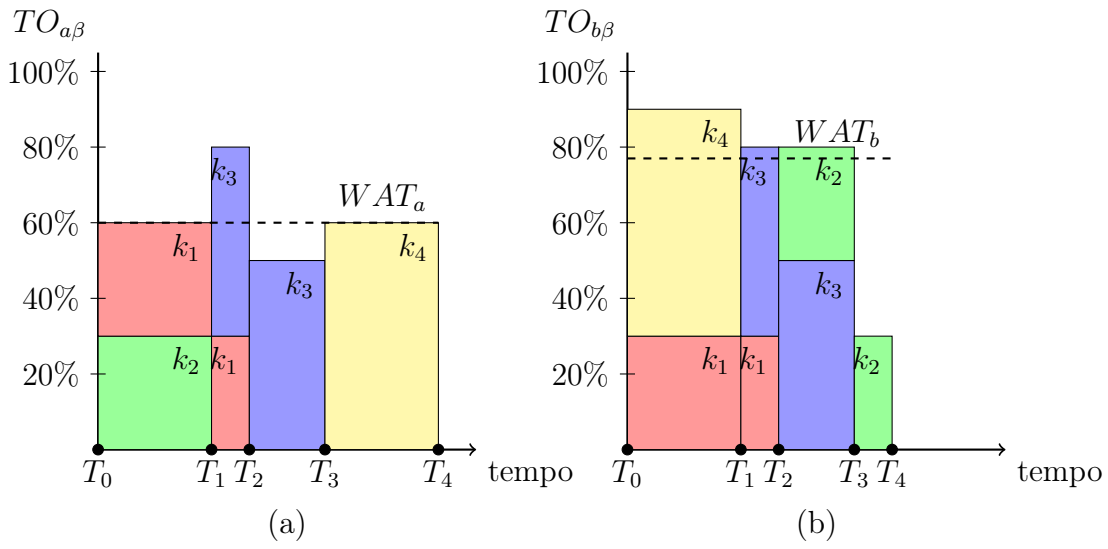

Figura 4.2: Taxa de ocupação $TO_{\alpha\beta}$ dos exemplos de ordenação Ord_α para cada instante T_β e também apresenta uma linha tracejada com a média ponderada da $TO_{\alpha\beta}$

um grupo de kernels que comporão a lista de execução e a cada término da execução de um kernel simulado, novos kernels serão chamados para serem executados. A ordenação dos kernels simula a execução dos mesmos de tal forma que o término de um e a entrada de outros kernels correspondem à ordem com que os kernels devem ser executados.

No inicio da execução da ordenação T_0, assume-se que todos os recursos dos dispositivos D chamado $W_j^{av} \mid j \in \{1, 2, 3\}$ estão disponíveis. Neste momento, será selecionado um conjunto de kernels para iniciar a simulação da execução.

Para um certo instante $T_\beta \mid \beta \in \{0, ..., \beta, ...N\}$ da execução da ordenação, um conjunto de kernels já executado, outros estão executando e o restante está aguardando a submissão para ser simulado. Neste instante, o kernel em execução que terminar primeiro, libera seus recursos incrementando no W_j^{av} e busca por outros kernels que ainda não foram submetidos.

A figura 4.3 ilustra o instante T_β para os dois exemplos de ordenação Ord_α. Na parte (a), o kernel k_2 libera os recursos do dispositivo D, atualizando W_j^{av} e busca por um conjunto de kernels que ocupe esses recursos W_j^{av}. Na parte (b), o kernel k_4 libera os recursos e disponibiliza os recursos para o kernel k_3 entrar em execução.

No instante T_N, todos os kernels terão sido simulados e enfileirados. Nesse momento, será construído uma lista seguindo a ordem das filas e seguindo o modelo *Round-Robin*. A figura 4.4 mostra o resultado da lista criada a partir do *Round-Robin* das filas construídas.

Neste trabalho, a escolha da submissão simulada dos novos kernels é feita através da

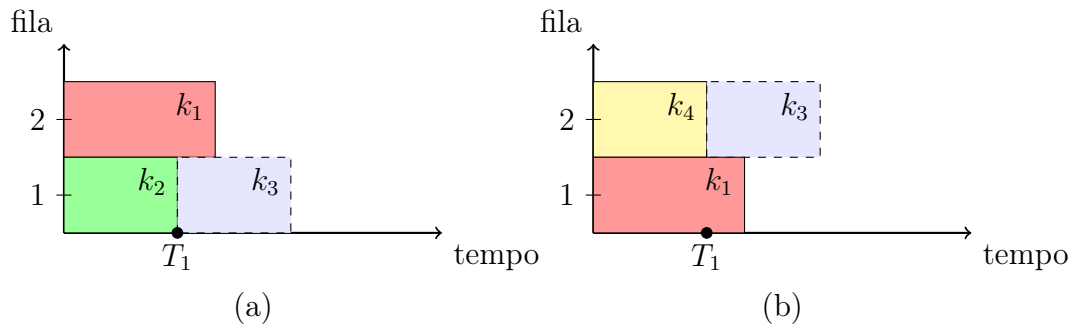

Figura 4.3: Instante T_i dos dois examples de ordenação

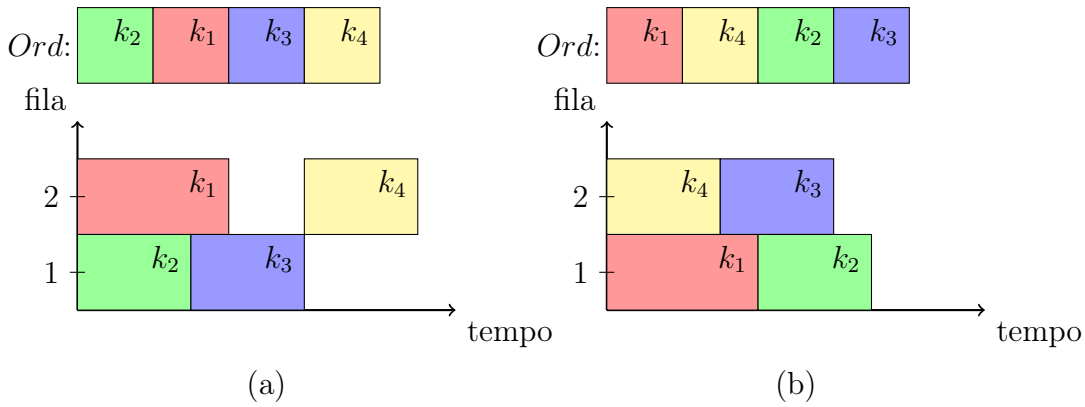

Figura 4.4: Instante T_N e a construção da lista utilizando o modo *Round-Robin* para cada ordenação vista anteriormente

solução do problema da mochila multidimensional, onde os itens da mochila representam os kernels e a capacidade da mochila representa os recursos disponíveis W_j^{av} do dispositivo. O problema da mochila precisa ser multidimensional porque um kernel possui diferentes tipos de recursos independentes e todos eles devem estar disponíveis para que o item seja alocado na mochila.

4.3 Problema da Mochila 0-1 Multidimensional (PMM)

O Problema da Mochila Multidimensional (PMM) [13][24] é um clássico problema de otimização combinatória que consiste em organizar N itens de diferentes pesos w_i e valores v_i dentro de uma mochila, objetivando maximizar os valores resultantes do somatório dos i itens selecionados e respeitar a capacidade máxima da mochila.

Este problema pode ser resolvido com Programação Dinâmica, obtendo a solução exata do problema, mas também pode ser resolvido com o Algoritmo Guloso e Meta Heurísticas para soluções aproximadas 4.6.

Considere N itens distintos de M dimensões e uma mochila com capacidade $W_j \mid j \in \{1, ..., M\}$. Seja $x_i \mid i \in \{1, ..., N\}$ a indicação do kernel k_i entrar na mochila quando $x_i = 1$ ou não quando $x_i = 0$. Cada item da mochila possui o peso $w_{ij} \mid i \in \{1, ..., N\}, j \in \{1, ..., M\}$ onde o somatório deles não podem exceder a capacidade da mochila seguindo a equação 4.4.

$$sujeito\ a \sum_{i=1}^{N} w_{ij} x_i \leq W_j \mid j \in \{1, ..., M\} \tag{4.4}$$

Cada item da mochila possui um valor $v_i \mid i \in \{1, ..., N\}$. O objetivo do problema é seguir a equação 4.5:

$$maximizar \sum_{i=1}^{N} v_i x_i \tag{4.5}$$

4.4 Adaptação do problema da mochila para o problema de reordenação de Kernels

O problema da mochila está sendo utilizado para selecionar kernels que irão ser processados concorrentemente a cada instante T_β. Para que um kernel entre em execução, o dispositivo D precisa ter memória compartilhada, registradores e threads suficientes, tornando o problema da mochila com dimensão $M = 3$. Assim, a equação 4.6 mostra o problema da mochila modificada para $M = 3$.

$$sujeito\ a \sum_{i=1}^{N} w_{ij} x_i \leq W_j \mid j \in \{1, 2, 3\} \tag{4.6}$$

A solução para o problema da mochila implementado com Programação Dinâmica representa uma solução ótima [24][23][5][41][3] para o somatório dos valores dos itens da mochila. Mapeando os valores dos itens da mochila através da equação 4.7 como a média das porcentagens dos recursos do item por unidade de tempo, o problema da mochila irá encontrar a solução ótima que maximiza o uso médio dos recursos por unidade de tempo.

$$v_i = \frac{\frac{w_{i1}}{W_1} + \frac{w_{i2}}{W_2} + \frac{w_{i3}}{W_3}}{3 \times t_i^{est}} \tag{4.7}$$

Para uma GPU com arquitetura Kepler, o número de filas de *hardware* é 32 ($NQ =$

32). Assim, um conjunto de filas Q serão utilizados para ordenar os kernels k_i. Os kernels são escalonados em cada fila usando o modo *Round-Robin* enquanto tiver recursos disponíveis.

4.5 Simplificação do Problema

Este trabalho modelou o problema da mochila de forma multidimensional. Porém, como a implementação do PMM é computacionalmente caro realizamos adaptações para torna-lo equivalente ao problema clássico multidimensional.

Portanto, ao invés de assumir a capacidade da GPU como sendo $W_j \mid j \in \{1, 2, 3\}$, a adaptação para o problema clássico da mochila fará com que a capacidade da GPU seja representado como descrito na equação 4.8 e os recursos w_{ij} necessários para o kernel k_i executar seja mapeado para equação 4.9.

$$W = (W_1 + 1) \times (W_2 + 1) \times (W_3 + 1) \tag{4.8}$$

$$w_i = (w_{i1} \times (W_2 + 1) \times (W_3 + 1)) + (w_{i2} \times (W_2 + 1)) + w_{i3} \tag{4.9}$$

4.6 Algoritmo para ordenação de kernels

O Algoritmo 1 descreve a proposta de ordenação cujos parâmetros de entrada são os kernels para serem ordenados e a quantidade de recursos disponíveis na GPU. A tabela 4.3 descreve todas as variáveis dos algoritmos desse capítulo. O Algoritmo consiste em um *loop* (linha 2) enquanto todos os kernels não foram submetidos, onde cada instante T_β é um iteração desse *loop*. No instante T_β, é solicitado para o procedimento $GetkernelsToSubmit$ (linha 3) que os kernels selecionados saiam da fila de espera e entrem em execução. Para cada kernel k_i selecionado (linha 4), será enfileirado os NQ kernels k_i na fila mais curta (linha 5 e 6), decrementado w_i dos recursos disponíveis W^{av} (linha 7) e removido o k_i da lista de kernels a serem submetidos (linha 8). Para finalizar o instante T_β, será buscado o kernel k_e que irá terminar antes(linha 9) e irá liberar os recursos w_e dos recursos disponíveis W^{av} (linha 10). No final de todas as iterações de T_β, as filas estarão preenchidas com os kernels selecionados e serão retornados os kernels na ordem das filas seguindo o modo *Round-Robin*.

Nome da variável	Descrição
$KernelList$	Lista de kernels a serem processados
$NextSubmitList$	Lista de kernels a serem escalonados nas filas
$SelectedKernels$	Conjunto de kernels selecionados
$SortedKernelList$	Lista de kernels ordenado
$tempW$	Variável auxiliar para armazenar a capacidade do dispositivo
W^{av}	Recursos totais disponíveis para o dispositivo
NQ	Número de filas de hardware
Q	Conjunto de filas de hardware
q_i	A fila de hardware de índice i
$q_i.first$	Primeiro elemento da fila de hardware q_i
$q_i.last$	Último elemento da fila de hardware q_i
k_i	Kernel de índice i
w_i	Recursos necessários para o kernel k_i
v_i	Valor associado ao kernel k_i
RRC	Lista de kernels seguindo o modo *Round-Robin*
M_{ij}	Matriz da programação dinâmica para o problema da mochila

Tabela 4.3: Variáveis utilizadas nos algoritmos

Algorithm 1 Algoritmo de ordenação dos Kernels

1: **function** KernelReorder($KernelList, W^{av}$)
2: **while** $KernelList$ not empty **do**
3: $NextSubmitList \leftarrow$ GetKernelsToSubmit($KernelList, W^{av}$)
4: **for each** $k_i \in NextSubmitList \wedge i \leq NQ$ **do**
5: $q_j \leftarrow$ FindShortQueue(Q)
6: $q_j \leftarrow q_j + k_i$
7: $W^{av} = W^{av} - w_i$
8: $KernelList = KernelList - k_i$
9: $q_j \leftarrow$ FindShortQueue(Q)
10: $k_e \leftarrow q_j.last$
11: $W^{av} = W^{av} + w_e$
12: **return** RoundRobingCross(Q)

A implementação dos procedimentos $FindShortQueue$ não serão detalhados nesse trabalho porque ele pode ser resolvida através de uma fila de prioridade [42][43] (*Heap*). Nesta fila de prioridade, os nós podem ser representados pela filas q_i, cujos valores representam o tempo total de execução de todos os kernels aguardando na fila. O procedimento $FindShortQueue$ irá usar dessa fila de prioridade para responder a fila mais curta, com menor valor, com um tempo de acesso $O(1)$ e um tempo de atualização $O(\log N)$, onde N é o número de kernels.

O Algoritmo 2 mostra a forma com é feito o *Round-Robin* das filas. A primeiro etapa do algoritmo é criar uma lista RRC vazia de kernels (linha 2). Depois, enquanto houver algum elemento na fila q_i (linha 7), será adicionado o primeiro kernel da espera q_i no RRC e removido o mesmo da fila (linha 9, 10 e 11).

Algorithm 2 Algoritmo de *Round-Robin* das filas

```
1: function RoundRobingCross(Q)
2:     RRC ← ∅
3:     f ← true
4:     while f do
5:         f ← false
6:         for each q_i ∈ Q do
7:             if q_i not empty then
8:                 f ← true
9:                 k_i ← q_i.first
10:                RRC ← RRC + k_i
11:                q_i ← q_i − k_i
12:    return RRC
```

O procedimento $GetKernelsToSubmit$ recebe como entrada os kernels que ainda não foram submetidos e a quantidade de recursos disponíveis no instante T_β. Esse procedimento irá implementar o problema da mochila com as adaptações descritas nesta seção utilizando o algoritmo de Programação Dinâmica e Guloso.

4.6.1 Algoritmo Guloso

O método Guloso [14] é uma técnica mais simples que pode ser aplicada a uma grande variedade de problemas. A grande maioria destes problemas possui um conjunto de entradas e qualquer subconjunto que satisfaça as restrição do problema, representa uma solução viável. Deseja-se então encontrar uma solução viável que maximize ou minimize uma dada função objetivo.

A implementação do Problema da Mochila utilizando o método Guloso é bem mais simples do que a Programação Dinâmica. O Algoritmo 3 implementa o procedimento utilizando a estratégia gulosa [17], onde o critério de ordenação é a razão entre o valor e o peso.

De forma simplificada, a implementação gulosa inicia ordenando a lista de kernels na ordem decrescente da razão entre o valor e o peso (linha 4). Depois disso, seguindo a ordem da ordenação, para cada kernel k_i (linha 5), se houver recursos suficientes $tempW$ para k_i (linha 6), os recursos w_i do kernel serão decrementados (linha 7) e ele será selecionado (linha 8).

Apesar da simplicidade da estratégia gulosa, a técnica não produz sempre soluções ótimas, existindo situações simples que o algoritmo é levado a escolher um item de maior relação entre valor e peso que não faz parte da solução ótima.

Algorithm 3 Implementação do *GetKernelsToSubmit* usando o Guloso como implementação do problema da mochila

```
function GetKernelsToSubmit(KernelList, W^av)
    SelectedKernels ← ∅
    tempW ← W^av
    SortedKernelList ← SortDecrementValuesPerWeight(KernelList)
    for each k_i ∈ SortedKernelList do
        if w_i < tempW then
            tempW ← tempW − w_i
            SelectedKernels ← SelectedKernels ∪ k_i
    NextSubmitList ← OrderSet(SelectedKernels)
    return NextSubmitList
```

4.6.2 Programação Dinâmica

Para implementar o problema da mochila com programação dinâmica, é preciso decompor o problema grande em problemas menores. Para isso, propomos usar uma fórmula recursiva para percorrer todas as possíveis decisões. A equação 4.10 [23] mostra a equação que denota o valor máximo obtido na mochila ao combinar os valores dos pesos w_i com a inclusão dos kernels através de x_i.

$$\begin{aligned} OPT(i,w) &= \max\left(\sum_{i=1}^{N} v_i x_i\right) \\ sujeito\ a &= \sum_{i=1}^{N} w_{ij} x_i \leq W_j \mid j \in \{1,2,3\} \end{aligned} \tag{4.10}$$

A partir da equação 4.10, o cálculo de recorrência para o valor máximo obtido na mochila pode ser expressa pela equação 4.11.

$$\begin{aligned} OPT(i,w) &= 0, i = 0 \\ &= OPT(i-1,w), w_i > w \\ &= \max(OPT(i-1,w), v_i + OPT(i-1, w-w_i)) \end{aligned} \tag{4.11}$$

Sabe-se que $OPT(i,w)$ é igual ao $OPT(i-1,w)$ caso o peso w_i do novo item seja maior do que a capacidade atual da mochila. Caso contrário, ele será igual ao máximo entre o estado atual sem o item $OPT(i-1,w)$ e o possível novo estado $OPT(i-1,w-w_i)+v_i$ onde é colocado o item i, reduzido a capacidade da mochila w_i e incrementado o valor v_i.

Com a equação de recorrência 4.11, iremos usar uma matriz M para armazenar e

consultar os resultados dos valores $OPT(i, w)$ anteriormente processados. O Algoritmo 4 implementa a criação da matriz M utilizando esta equação de recorrência.

Algorithm 4 Criação da Matrix a partir da Equação de Recorrência

```
 1: function KnapsackDynamicMatrix(KernelList, W^av)
 2:     M[N + 1, W + 1]
 3:     for j = 0 to W do
 4:         M[0, j] ← 0
 5:     for i = 1 to N do
 6:         for j = 1 to W do
 7:             if (w_i > j) then
 8:                 M[i, j] ← M[i − 1, j]
 9:             else
10:                 M[i, j] ← max(M[i − 1, j], v_i + M[i − 1, j − w_i])
11:     return M
```

Com a matriz M construída, os itens selecionados na mochila que maximizam os valores podem ser recuperados. Então, o procedimento $GetKernelsToSubmit$ apresentado no Algoritmo 5 usará a matriz M para recuperar os kernels k_i selecionados e retorná-los na ordem decrescente de valor.

O Algoritmo 5 inicia o conjunto dos kernels selecionados como vazio (linha 2). Ao longo da execução, a quantidade de memória disponível será decrementada toda vez que se selecionar um kernel k_i (linha 8). Para isso, foi criada uma variável auxiliar $tempW$ que indica a quantidade de memória disponível a cada iteração. O algoritmo cria a matriz M através do procedimento $KnapsackDynamicMatrix$ e percorre do último kernel k_i até o primeiro (linha 5) verificando se ainda há memória disponível (linha 6) e se o valor da célula da matriz $M[i][tempW] \neq M[i-1][tempW]$, significando que o kernel k_i foi selecionado (linha 7). Quando a seleção de um kernel k_i ocorre, será decrementados os recursos disponíveis de w_i (linha 8) e será incluído o k_i no conjunto de kernels selecionados.

4.6.3 Exemplo da construção de Matriz

Considere um exemplo de cinco kernels com uma GPU que tem $NSM = 1$ e com capacidade $W_1 = 2$, $W_2 = 3$ e $W_3 = 4$ gerando $W = 60$ através da equação 4.8. Os recursos necessários dos kernels k_i são apresentados na tabela 4.4 com seus respectivos valores de w_i através da equação 4.9 e valores v_i vindo da equação 4.7.

A matriz M pode ser construída através da equação de recorrência 4.11 de Programação Dinâmica para resolver o Problema da Mochila. A tabela 4.5 mostra um exemplo de uma matriz M, onde pode ser observado que cada linha representa um kernel k_i e cada coluna representa a quantidade linearizada dos recursos w_i. Por exemplo, uma coluna 58

k_i	$\overrightarrow{w_i}$	w_i	v_i
k_1	$(1,1,1)$	26	20
k_2	$(1,2,3)$	33	40
k_3	$(1,0,2)$	22	30
k_4	$(1,3,1)$	36	60
k_5	$(2,1,1)$	51	70

Tabela 4.4: Valores de w_i e v_i para cada kernel k_i

indica a quantidade $w_i = 58$ ou $W_1 = 2$, $W_2 = 3$ e $W_3 = 3$.

De acordo com a equação 4.11, a célula $M[k_i, w_j]$ pode ser atualizado com o maior valor entre $M[k_{i-1}, j - w_i] + v_i$ e $M[k_{i-1}, j]$.

O primeiro caso ilustrado na tabela 4.5 ocorre quando o objeto selecionado aumenta o lucro da mochila. Isto acontece, por exemplo, na célula $M(k_4, 58)$, onde $v_4 = 60$, $w_4 = 36$ e $M[k_3, 58 - 36] + 60 > M[k_3, 58]$. Nesse caso, o valor da célula $M(k_4, 58)$ é atualizada para $M[k_3, 22] + 60 = 90$ em função do aumento do lucro da mochila ao adicionar o objeto.

$OPT(i,w)$	0	1	2	...	7	...	22	...	57	58	59
0	0	0	0	...	0	...	0	...	0	0	0
k_1	0	0	0	...	0	...	0	...	20	20	20
k_2	0	0	0	...	0	...	0	...	40	40	60
k_3	0	0	0	...	0	...	30	...	70	70	70
k_4	0	0	0	...	0	...	30	...	70	?	?
k_5	?	?	?	...	?	...	?	...	?	?	?

Tabela 4.5: Calculo do valor de $OPT(k_4, 58)$

O segundo caso ilustrado na tabela 4.6 ocorre quando um objeto não acrescenta lucro na mochila. Isto pode ser ilustrado quando a célula $M(k_5, 58)$ for calculada, onde $M[k_4, 58] > M[k_4, 58 - 51] + 70$. Para esse caso, o valor da célula $M[k_5, 58]$ é atualizada para $M[k_4, 58] = 90$.

$OPT(i,w)$	0	1	2	...	7	...	22	...	57	58	59
0	0	0	0	...	0	...	0	...	0	0	0
k_1	0	0	0	...	0	...	0	...	20	20	20
k_2	0	0	0	...	0	...	0	...	40	40	60
k_3	0	0	0	...	0	...	30	...	70	70	70
k_4	0	0	0	...	0	...	30	...	70	90	90
k_5	0	0	0	...	0	...	0	...	70	90	?

Tabela 4.6: Calculo do valor de $OPT(k_5, 59)$

Algorithm 5 Implementação do $GetKernelsToSubmit$ usando a Programação Dinâmica como implementação do problema da mochila

```
1: function GetKernelsToSubmit(KernelList, W^av)
2:     SelectedKernels ← ∅
3:     tempW ← W^av
4:     M ← KnapsackDynamicMatrix(KernelList, W^av)
5:     for i ← N to 1 do
6:         if tempW > 0 then
7:             if M[i][tempW] ≠ M[i − 1][tempW] then
8:                 tempW ← tempW − w_i
9:                 SelectedKernels ← SelectedKernels ∪ k_i
10:    NextSubmitList ← OrderSet(SelectedKernels)
11:    return NextSubmitList
```

4.6.4 Vantagens e Desvantagens do Guloso e da Programação Dinâmica

A implementação do problema da mochila usando Programação Dinâmica que prove a solução ótima, enquanto que usando o método Guloso produz uma boa solução apresentam assim vantagens e desvantagens para cada método.

A Programação Dinâmica é muito utilizada quando a diferença entre uma solução boa e ótima é impactante para a aplicação. Esse método garante a solução ótima para o problema, porque leva em consideração o contexto global, mas requer a construção e o preenchimento de uma matriz M cuja a dimensão é o número de itens a ser colocados na mochila. Dependendo da quantidade de itens e da capacidade da mochila, o método pode se tornar inviável.

Em função da simplicidade do método Guloso, a complexidade de processamento e de memória não costuma ser um problema. A heurística utilizada na ordenação dos itens usa da razão entre o valor e o peso para representar uma boa solução num contexto local, não gerando o melhor resultado para o problema.

Capítulo 5

Resultados Experimentais

5.1 Ambiente Computacional

Para fazer o levantamente dos resultados, foi utilizado uma TitanX e uma K40 para executar os testes. A tabela 5.1 mostra as duas GPUs que foram utilizadas nos testes. Todos os kernels implementados neste trabalho foram implementados em CUDA 7.5. Para criar os *scripts* de testes e gerenciar os resultados foi utilizada a linguagem Java 1.7 .

Nesta tabela pode ser notado que as duas GPUs possuem arquiteturas diferentes, onde a K40 possui a arquitetura Kepler e a TitanX possui arquitetura Maxwell. A utilização de duas GPUs com arquiteturas diferentes permitirá mostrar a portabilidade da solução.

	TitanX	**K40**
Número de Cores	3,072	2,880
Core Clock	1000 MHz	745 MHz
RAM	24GB	12GB
Memory Bandwidth	336.5 GB/s	288 GB/s
Capability	5.2	3.5
Número de SMs (NSM)	24	15
Shared Memory por SM (R_1)	96KB	48KB
Numero of Registers por SM (R_2)	64K	64K
Max número de threads por SM (R_3)	2048	2048
Arquitetura	Maxwell	Kepler
Pico de desempenho	6.6 TFLOPS	4.2 TFLOPS

Tabela 5.1: Configuração das GPUs nos resultados

5.2 Metodologia

A execução de kernels concorrentes é uma funcionalidade razoavelmente nova em GPUs da NVIDIA, disponível de forma rudimentar desde a arquitetura Fermi. Os programadores CUDA estão mais acostumados a explorar os ganhos do paralelismo entre *threads* de um mesmo kernel ao invés da exploração do paralelismo entre kernels. Esse trabalho considera cenários onde kernels gerenciados pela GPU são executados de forma concorrente, podendo ou não ser de uma mesma aplicação.

Existem diversos *benchmarks* para CUDA na literatura, tais como Rodinia [8], Parboil [39] e CUDA SDK [27], compostos por conjuntos de aplicações de kernels independentes. Essas aplicações requerem diferentes quantidades de recursos, que representam uma boa amostragem de kernels com diferentes usos de recursos. Em função da complexidade dos algoritmos dos kernels encontrados nestes *benchmarks* e da necessidade de um número muito grande de kernels diferentes, este trabalho decidiu criar um benchmark de kernels sintéticos, possibilitando um controle refinado do número de kernels e de suas especificações.

Este trabalho implementa um gerador de kernels que cria um conjunto de N kernels independentes com os recursos $R_j \mid j \in \{1, 2, 3\}$. Os valores de N e R_j são passados como parâmetro para o gerador. A estrutura geral de cada kernel k_i sintético consiste num conjunto de códigos CUDA que realizam algumas operações aritméticas em um vetor de inteiros alocado na memória compartilhada. Uma quantidade de registradores também é alocado, mas os resultados mostraram que os registradores influenciam pouco na concorrência entre os kernels. Com isso, o gerador cria kernels com um número mínimo de registradores para diminuir a complexidade do processamento da rereordenação. O valor do tempo estimado de cada kernel pode ser determinado em função da quantidade de vezes que um *loop* interno do kernel é executado.

Neste trabalho, foi criado um conjunto distinto de kernels de tamanho variados, onde N pode ter os seguintes valores 32, 64, 128 ou 256. Em cada conjunto, foi também variado o número máximo e mínimo de blocos criados para cada kernel. A quantidade mínima de blocos é sempre 4 e a quantidade máxima NB pode ser 32, 64, 128, 256, ou 512. Todos os números atribuídos no experimento vem de um gerador aleatório. Para cada experimento com N kernels, foram gerados 50 diferentes conjuntos de kernels com valores aleatórios de blocos e recursos para minimizar o efeito da aleatoriedade dos testes.

Neste trabalho foram utilizadas as métricas *Average Normalized Turnaround Time*

($ANTT$) e *System Throughput* (STP) para avaliar os resultados dos experimentos.

A métrica $ANTT$, proposta por Eyerman [12][46], é calculada pela equação 5.1 e representa a média normalizada da razão entre o TT_i e o TT_i^{Alone} de todos os kernels para uma reordenação, onde TT_i^{Alone} representa o valor do *turnaround time* da execução do k_i quando o kernel é executado sozinho e TT_i representa o valor do *turnaround time* da execução do kernel k_i executando concorrentemente com outros kernels. Em outras palavras, essa métrica indica em média o quanto o kernel k_i foi prejudicado pela concorrência com relação a execução de um ambiente não concorrente. Nesta métrica o valor deve estar no intervalo dado por $\{1, ..., N\}$ sendo que quanto menor, melhor, onde o valor 1 indica que a concorrência não interferiu em nada na execução dos kernels e N indicando que os kernels estão em total conflito de acesso a recursos.

$$ANTT = \frac{1}{N} \sum_{i=1}^{N} \frac{TT_i}{TT_i^{Alone}} \tag{5.1}$$

A métrica STP, também proposta por Eyerman [12][46], é calculada pela equação 5.2 e representa o somatório da fração dos TT_i^{Alone} com os TT_i para cada reordenação. Nesta métrica o valor deve estar no intervalo de $\{1, ..., N\}$ e o quanto maior, melhor.

$$STP = \sum_{i=1}^{N} \frac{TT_i^{Alone}}{TT_i} \tag{5.2}$$

5.3 Desempenho

Este trabalho implementou duas técnicas de reordenação de kernels utilizando o Problema da Mochila com o método Guloso (*Greedy*) e Programação Dinâmica (*Dynamic*). Os resultados das reordenação desses dois métodos serão comparados com o resultado dado pela reordenação do programa (*Standard*), onde os N kernels são reordenados na ordem crescente pelo índice. Por exemplo, para $N = 4$, a reordenação *Standard* corresponderá a $Ord = \{k_1, k_2, k_3, k_4\}$.

As figuras 5.1 a 5.3 mostram o *profile* da execução dos kernels reordenados testados na TitanX. As figuras mostram as filas de tarefas no eixo y e a linha de tempo no eixo x com uma amostra de 2 segundos, onde cada caixa retangular do *trace* representa a execução de um kernel. Pode-se observar que muitos kernels, representados por caixas retangulares, são executados concorrentementes no inicio da figura 5.3 e poucos são executados na figura

5.1. Em ambos os testes, a GPU suporta até 32 filas de trabalhos simultâneos onde o *trace* da reordenação utilizou de até 13 filas para o *Standard*, 17 para *Greedy* e 20 para *Dynamic*.

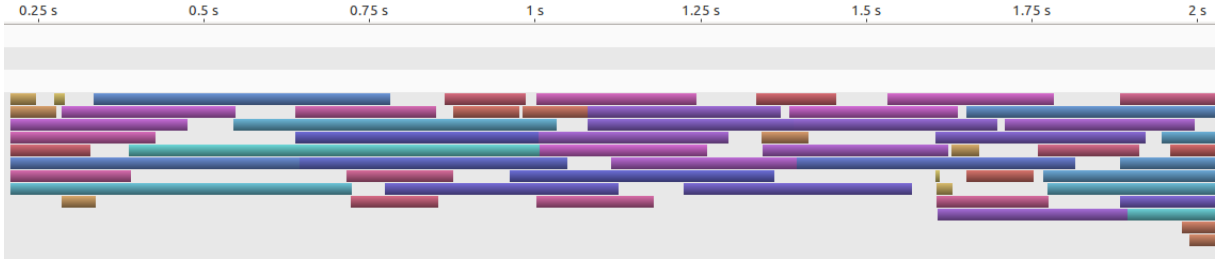

Figura 5.1: *Trace* de execução da ordenação *Standard* com 64 kernels

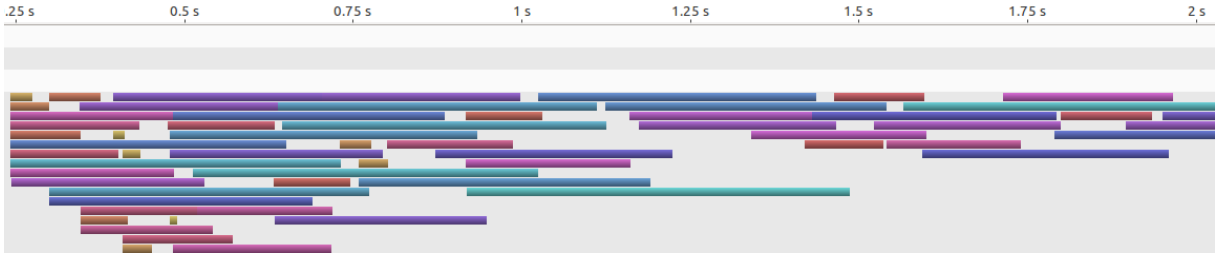

Figura 5.2: *Trace* de execução da ordenação *Greedy* com 64 kernels

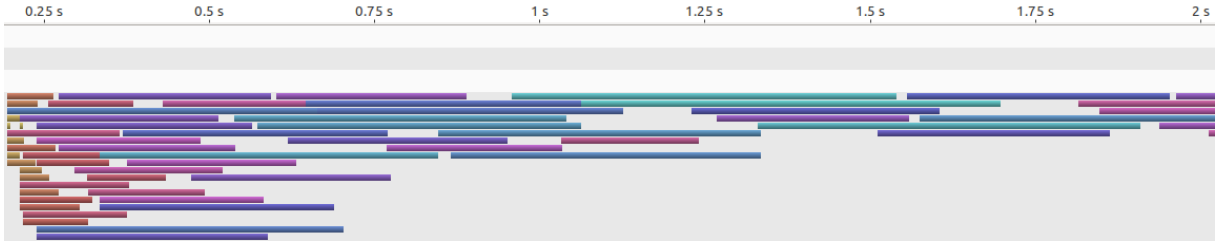

Figura 5.3: *Trace* de execução da ordenação *Dynamic* com 64 kernels

5.3.1 Average Normalized Turnaround Time

As figuras 5.4 e 5.5 mostram os resultados dos valores do *ANTT* para diferentes configurações de execução, onde são alterados a quantidade de número de kernels N e número de blocos NB.

Nestes resultados, o ganho do *Greedy* em relação ao *Standard* é de 17% a 70% e o ganho do *Dynamic* perante o *Standard* é de 55% a 86%. Pode ser observado nas figuras que a diferença de valores da reordenação *Standard* com o *Dynamic* é significativa.

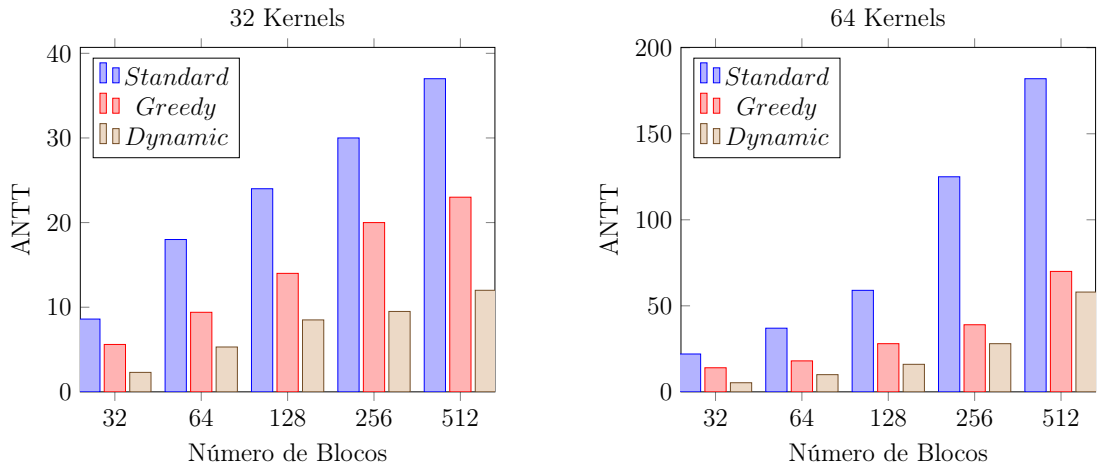

Figura 5.4: Resultado do ANTT para $N = 32$ e $N = 64$ na TitanX.

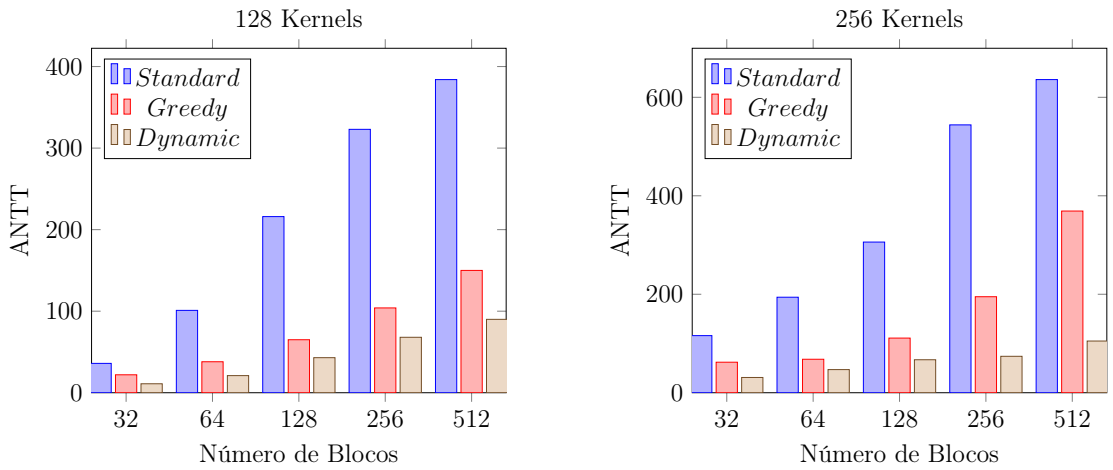

Figura 5.5: Resultado do ANTT para $N = 128$ e $N = 256$ na TitanX.

5.3.2 System Throughput

As figuras 5.6 e 5.7 mostram os valores do resultado do STP para as mesmas configurações dos resultados do $ANTT$, onde são variados a quantidade de número de kernels N e número de blocos NB.

Nestes resultados, o ganho do *Greedy* perante o *Standard* é de 27% a 58% e o ganho do *Dynamic* perante o *Standard* é de 43% a 67%. Pode ser observado nas figuras que a diferença de valores da reordenação *Standard* com o *Dynamic* também é significativo.

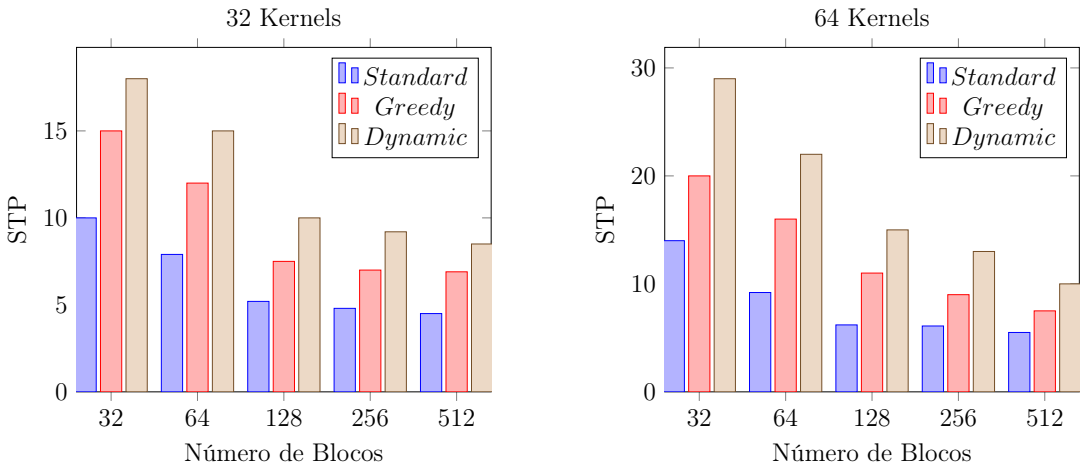

Figura 5.6: Resultado do STP para $N = 32$ e $N = 64$ na TitanX.

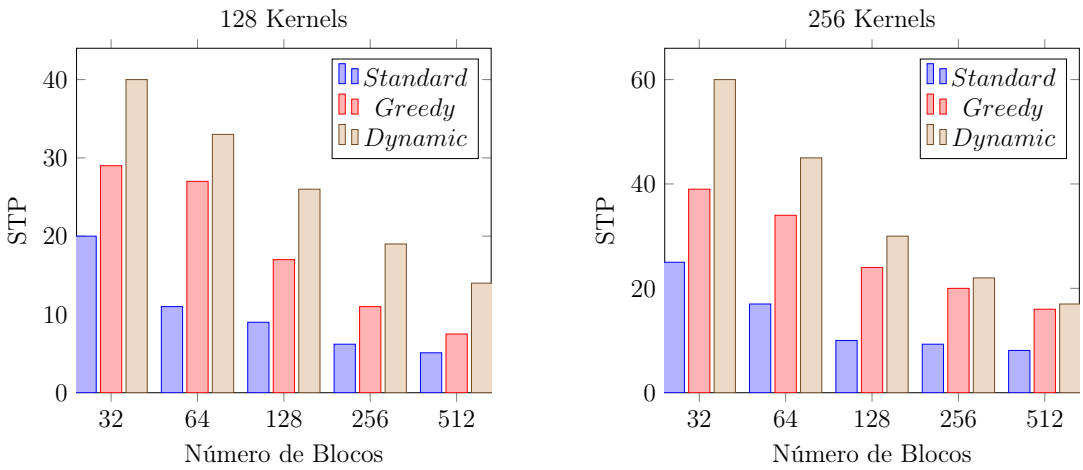

Figura 5.7: Resultado do STP para $N = 128$ e $N = 256$ na TitanX.

5.4 Custo de execução do método proposto

O algoritmo de reordenação é realizado estaticamente antes de executar os kernels. Cada reordenação possui um tempo de processamento e dependendo do seu tamanho pode representar um gargalo. Por exemplo, a implementação do problema da mochila com Programação Dinâmica requer a construção de uma matriz com uma dimensão considerável. Nos testes realizados neste trabalho, a diferença do tempo da reordenação *Greedy* para o *Dynamic* é significativo. O custo de execução de uma reordenação chamado de *overhead* representa a razão do tempo de processamento com a diferença do *Average Turnaround Time* (*ATT*) entre a reordenação e o *Standard*. A equação 5.3 mostra a formula do *ATT* e a equação 5.4 mostra a fórmula do custo ou *overhead* O.

$$ATT_\alpha = \frac{\sum_{i=1}^{N} TT_i}{N} \mid \alpha \in \{greedy, dynamic\} \tag{5.3}$$

$$O_\alpha = \frac{OT_\alpha \times ANTT_{standard}}{(ANTT_{standard} - ANTT_\alpha)} \mid \alpha \in \{greedy, dynamic\} \tag{5.4}$$

As tabelas 5.2 e 5.3 mostram os dois tipos de reordenação com suas respectivas diferença de *overhead*. Para o *overhead* do *Dynamic*, em trabalhos futuros e contando com o aumento do número de recursos da GPU, o tamanho da matriz tende a aumentar fazendo com que o $OT_{dynamic}$ da reordenação fique pior. Para que o *overhead* se preserve, a proporção deverá se manter e o valor do ganho entre o $ATT_{dynamic}$ e $ATT_{standard}$ deverá aumentar para compensar o aumento do tempo de processamento.

O_{greedy}	Blocos				
	32	64	128	256	512
32 Kernels	0.024	0.008	0.002	0.002	0.001
64 Kernels	0.036	0.005	0.002	0.001	0.009
128 Kernels	0.057	0.006	0.002	0.001	0.002
256 Kernels	0.024	0.008	0.002	0.002	0.001

Tabela 5.2: *Overhead* O_{greedy} do Guloso na TitanX

$O_{dynamic}$	Blocos				
	32	64	128	256	512
32 Kernels	4.2	2.1	1.0	0.9	0.6
64 Kernels	3.6	2.1	1.5	1.2	1.1
128 Kernels	4.1	2.9	2.1	1.6	1.0
256 Kernels	6.3	5.4	3.7	2.6	1.9

Tabela 5.3: *Overhead* $O_{dynamic}$ da Programaçã Dinâmica na TitanX

5.5 Portabilidade do método entre diferentes arquiteturas

Uma solução de software foi utilizada para melhorar o *turnaround time* dos kernels quando executados concorrentemente. Essa mesma técnica pode ser aplicada em diferentes arquiteturas que possuem a tecnologia Hyper-Q da NVIDIA. Os resultados apresentados são especialmente significativos com as arquiteturas mais recentes, tais como a Kepler e Maxwell.

5.5.1 Average Normalized Turnaround Time

As figuras 5.8 e 5.9 mostram os resultados dos valores do $ANTT$ para uma placa $K40$ (Kepler). Pode ser observado pelas figuras que o padrão de comportamento da arquitetura Kepler se mantém semelhante a Maxwell.

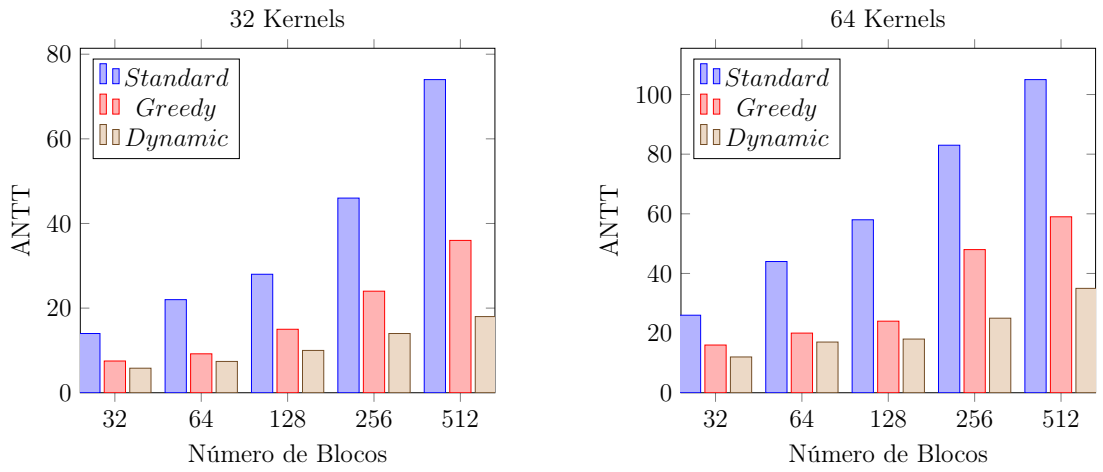

Figura 5.8: Resultado do ANTT na K40 para $N = 32$ e $N = 64$

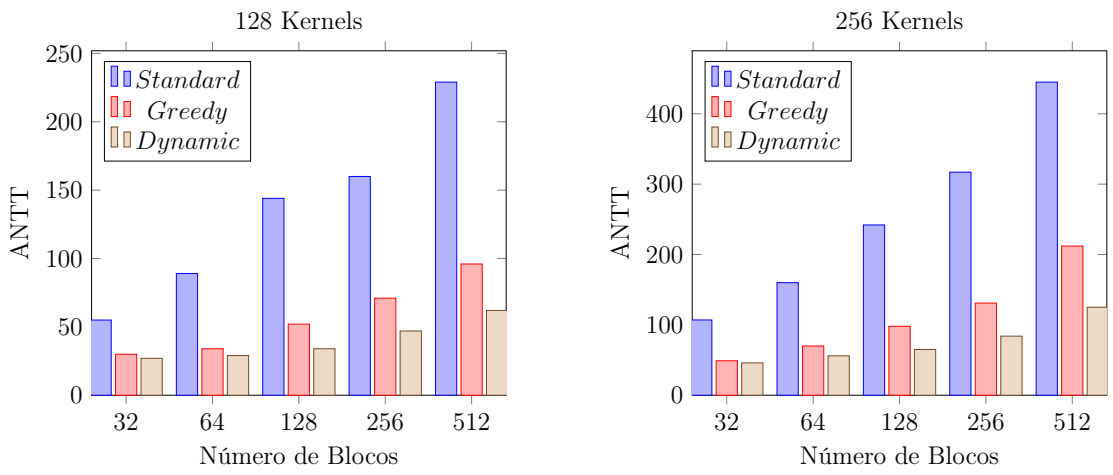

Figura 5.9: Resultado do ANTT na K40 para $N = 128$ e $N = 256$

5.5.2 System Throughput

As figuras 5.10 e 5.11 mostram os valores do $ANTT$ para uma placa $K40$ (Kepler). Comparando essas figuras com a da arquitetura Maxwell pode ser observado também uma semelhança no padrão de comportamento.

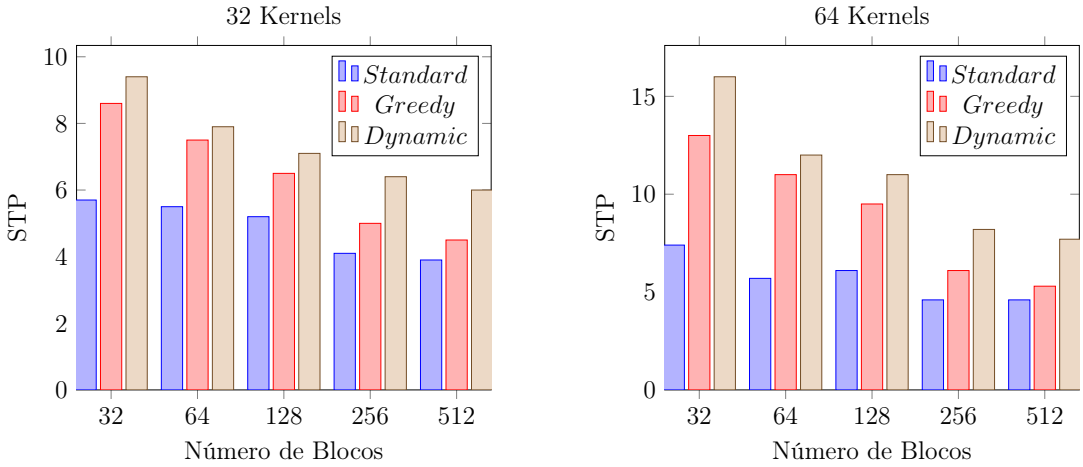

Figura 5.10: Resultado do STP na K40 para $N = 32$ e $N = 64$

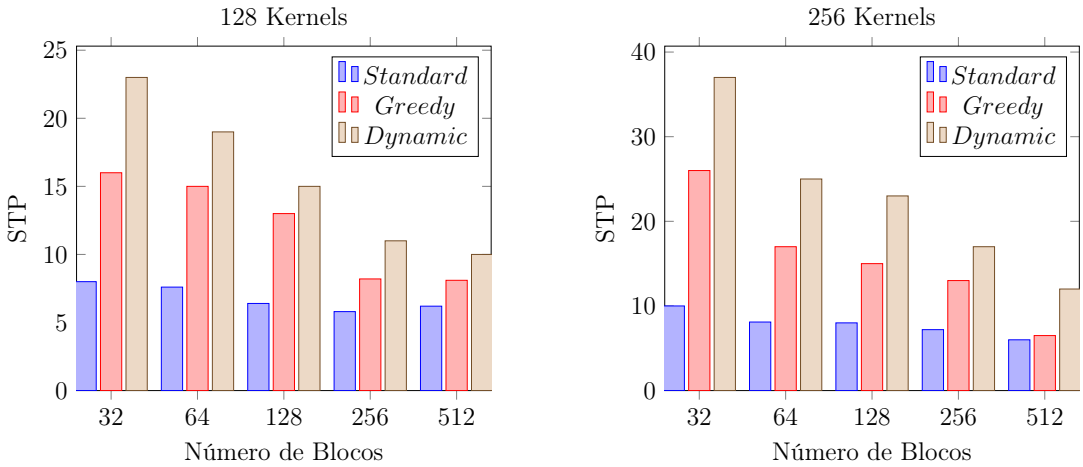

Figura 5.11: Resultado do STP na K40 para $N = 128$ e $N = 256$

Capítulo 6

Conclusões

Embora inicialmente as arquiteturas de GPUs possuíam unidades de controles bastante restritivas, permitindo apenas o seu uso com kernels sequenciais, o avanço nas unidades de controles dos SMs, incluindo recursos como o Hyper-Q, possibilitaram que kernels concorrentes possam ser executados.

Com esse cenário, o compartilhamento do uso dos recursos da GPU e com a crescente necessidade dos projetos em utilizar desse co-processador para efetuar grandes operações matemáticas, torna-se necessário aprofundar nos estudos e análises de como esta concorrência é realizada e como aproveitar melhor a taxa de ocupação do hardware nestas situações.

Neste trabalho apresentamos uma estratégia de ordenação via software que foca em melhorar o *turnaround time* médio dos kernels submetidos para a GPU. Para tanto, utilizamos técnicas de otimização baseadas no problema da mochila para encontrar uma ordem de execução dos kernels de tal forma a melhorar a utilização da GPU. Toda vez que um kernel termina e libera recursos para o dispositivo, novos kernels que podem ocupar melhor os espaços e recursos disponíveis são alocados.

Duas soluções do problema da mochila foram implementadas, com diferentes propostas e resultados, utilizando arquiteturas diferentes de GPUs. As mesmas foram testadas para verificar o quanto o resultado de uma solução é diferente da outra nas duas arquiteturas. Cada experimento usou diferentes números de kernels e blocos, e cada kernel precisa de um número aleatório de recursos.

Os resultados da proposta mostram um ganho de $ANTT$ do *Greedy* com *Standard* de 17% a 70% e do *Dynamic* com o *Standard* de 55% a 86% e um ganho do STP do *Greedy* com *Standard* de 27% a 58% e do ganho do *Dynamic* com o *Standard* de 43%

a 67%. Além disso, o *overhead* da ordenação é disperzível comparado ao ganho dessas métricas. O trabalho também mostra que a proposta possui um ganho similar tanto na arquitetura Kepler quanto na Maxwell.

Referências

[1] Adriaens, J. T.; Compton, K.; Kim, N. S.; Schulte, M. J. The case for gpgpu spatial multitasking. In *IEEE 18th International Symposium on High Performance Computer Architecture (HPCA)* (2012), IEEE, pp. 1–12.

[2] Akbar, M. M.; Manning, E. G.; Shoja, G. C.; Khan, S. Heuristic solutions for the multiple-choice multi-dimension knapsack problem. In *International Conference on Computational Science* (2001), Springer, pp. 659–668.

[3] Andonov, R.; Poirriez, V.; Rajopadhye, S. Unbounded knapsack problem: Dynamic programming revisited. *European Journal of Operational Research 123*, 2 (2000), 394–407.

[4] Andonov, R.; Raimbault, F.; Quinton, P. *Dynamic programming parallel implementations for the knapsack problem.* Tese de Doutorado, INRIA, 1993.

[5] Bertsimas, D.; Demir, R. An approximate dynamic programming approach to multidimensional knapsack problems. *Management Science 48*, 4 (2002), 550–565.

[6] Bradley, T. Hyper-q example. *NVidia Corporation. Whitepaper v1.0* (2012).

[7] Cassagne, A.; George, A.; Lorendeau, B.; Papin, J.-C.; Rougier, A. Concurrent kernel execution on graphic processing units. *no published* (2013).

[8] Che, S.; Boyer, M.; Meng, J.; Tarjan, D.; Sheaffer, J. W.; Lee, S.-H.; Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on* (2009), IEEE, pp. 44–54.

[9] Che, S.; Boyer, M.; Meng, J.; Tarjan, D.; Sheaffer, J. W.; Skadron, K. A performance study of general-purpose applications on graphics processors using cuda. *Journal of parallel and distributed computing 68*, 10 (2008), 1370–1380.

[10] Clua, E. W. G.; Zamith, M. P. Programming in cuda for kepler and maxwell architecture. *Revista de Informática Teórica e Aplicada 22*, 2 (2015), 233–257.

[11] CUDATM, N. Nvidia cuda c programming guide. *NVIDIA Corporation 2701* (2011).

[12] Eyerman, S.; Eeckhout, L. System-level performance metrics for multiprogram workloads. *IEEE micro 28*, 3 (2008), 42–53.

[13] Fréville, A. The multidimensional 0–1 knapsack problem: An overview. *European Journal of Operational Research 155*, 1 (2004), 1–21.

[14] Gorski, J.; Paquete, L.; Pedrosa, F. Greedy algorithms for a class of knapsack problems with binary weights. *Computers & Operations Research 39*, 3 (2012), 498–511.

[15] Gregg, C.; Dorn, J.; Hazelwood, K.; Skadron, K. Fine-grained resource sharing for concurrent gpgpu kernels. In *Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism* (2012).

[16] Guevara, M.; Gregg, C.; Hazelwood, K.; Skadron, K. Enabling task parallelism in the cuda scheduler. In *Workshop on Programming Models for Emerging Architectures* (2009), vol. 9.

[17] Horowitz, E.; Sahni, S. *Fundamentals of computer algorithms*. Computer Science Press, 1978.

[18] Jiao, Q.; Lu, M.; Huynh, H. P.; Mitra, T. Improving gpgpu energy-efficiency through concurrent kernel execution and dvfs. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (2015), IEEE, pp. 1–11.

[19] Karunadasa, N.; Ranasinghe, D. Accelerating high performance applications with cuda and mpi. In *2009 International Conference on Industrial and Information Systems (ICIIS)* (2009), IEEE, pp. 331–336.

[20] Li, T.; Narayana, V. K.; El-Ghazawi, T. A power-aware symbiotic scheduling algorithm for concurrent gpu kernels. In *Parallel and Distributed Systems (ICPADS), 2015 IEEE 21st International Conference on* (2015), IEEE, pp. 562–569.

[21] Liang, Y.; Huynh, H. P.; Rupnow, K.; Goh, R. S. M.; Chen, D. Efficient gpu spatial-temporal multitasking. *IEEE Transactions on Parallel and Distributed Systems 26*, 3 (2015), 748–760.

[22] Lopez-Novoa, U.; Mendiburu, A.; Miguel-Alonso, J. A survey of performance modeling and simulation techniques for accelerator-based computing. *IEEE Transactions on Parallel and Distributed Systems 26*, 1 (2015), 272–281.

[23] Martello, S.; Pisinger, D.; Toth, P. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science 45*, 3 (1999), 414–424.

[24] Martello, S.; Toth, P. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[25] Nickolls, J.; Buck, I.; Garland, M.; Skadron, K. Scalable parallel programming with cuda. *Queue 6*, 2 (2008), 40–53.

[26] NVIDIA. Cuda c programming guide. Disponível em `http://docs.nvidia.com/cuda/cuda-c-programming-guide`.

[27] NVIDIA. Cuda sdk. Disponível em `https://developer.nvidia.com/cuda-downloads`.

[28] NVIDIA. Fermi compute architecture white paper. White Paper, 2010.

[29] NVIDIA. Profiler :: Cuda toolkit documentation - nvidia documentation, 2011. Disponível em `http://docs.nvidia.com/cuda/profiler-users-guide/`.

[30] NVIDIA. Kepler gk110 compute architecture white paper. White Paper, 2012.

[31] NVIDIA. Nvidia kepler compute architecture, 2012. Disponível em `http://www.nvidia.com/object/nvidia-kepler.html`.

[32] NVIDIA. Maxwell architecture, 2014. Disponível em `https://developer.nvidia.com/maxwell-compute-architecture`.

[33] Owens, J. D.; Luebke, D.; Govindaraju, N.; Harris, M.; Krüger, J.; Lefohn, A. E.; Purcell, T. J. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum* (2007), vol. 26, Wiley Online Library, pp. 80–113.

[34] Pai, S.; Thazhuthaveetil, M. J.; Govindarajan, R. Improving gpgpu concurrency with elastic kernels. *ACM SIGPLAN Notices 48*, 4 (2013), 407–418.

[35] Park, J. J. K.; Park, Y.; Mahlke, S. Chimera: Collaborative preemption for multitasking on a shared gpu. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (2015), ACM, pp. 593–606.

[36] Peters, H.; Köper, M.; Luttenberger, N. Efficiently using a cuda-enabled gpu as shared resource. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on* (2010), IEEE, pp. 1122–1127.

[37] Ravi, V. T.; Becchi, M.; Agrawal, G.; Chakradhar, S. Supporting gpu sharing in cloud environments with a transparent runtime consolidation framework. In *Proceedings of the 20th international symposium on High performance distributed computing* (2011), ACM, pp. 217–228.

[38] Shahriar, A. Z. M.; Akbar, M. M.; Rahman, M. S.; Newton, M. A. H. A multiprocessor based heuristic for multi-dimensional multiple-choice knapsack problem. *The Journal of Supercomputing 43*, 3 (2008), 257–280.

[39] Stratton, J. A.; Rodrigues, C.; Sung, I.-J.; Obeid, N.; Chang, L.-W.; Anssari, N.; Liu, G. D.; Hwu, W.-m. W. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing 127* (2012).

[40] Tanasic, I.; Gelado, I.; Cabezas, J.; Ramirez, A.; Navarro, N.; Valero, M. Enabling preemptive multiprogramming on gpus. In *ACM SIGARCH Computer Architecture News* (2014), vol. 42, IEEE Press, pp. 193–204.

[41] Toth, P. Dynamic programming algorithms for the zero-one knapsack problem. *Computing 25*, 1 (1980), 29–45.

[42] van Emde Boas, P.; Kaas, R.; Zijlstra, E. Design and implementation of an efficient priority queue. *Mathematical Systems Theory 10*, 1 (1976), 99–127.

[43] Vuillemin, J. A data structure for manipulating priority queues. *Communications of the ACM 21*, 4 (1978), 309–315.

[44] Wang, G.; Lin, Y.; Yi, W. Kernel fusion: An effective method for better power efficiency on multithreaded gpu. In *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on & Int'l Conference on Cyber, Physical and Social Computing (CPSCom)* (2010), IEEE, pp. 344–350.

[45] Wang, L.; Huang, M.; El-Ghazawi, T. Exploiting concurrent kernel execution on graphic processing units. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on* (2011), IEEE, pp. 24–32.

[46] Wen, Y.; Wang, Z.; O'Boyle, M. F. Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms. In *2014 21st International Conference on High Performance Computing (HiPC)* (2014), IEEE, pp. 1–10.

[47] Wende, F.; Cordes, F.; Steinke, T. On improving the performance of multithreaded cuda applications with concurrent kernel execution by kernel reordering. In *Application Accelerators in High Performance Computing (SAAHPC), 2012 Symposium on* (2012), IEEE, pp. 74–83.

[48] Zanotto, L.; Ferreira, A.; Matsumoto, M. Arquitetura e programaçãoo de gpu nvidia. *no published* (2012).

[49] Zhong, J.; He, B. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems 25*, 6 (2014), 1522–1532.

[50] Zhu, Y.; Magklis, G.; Scott, M. L.; Ding, C.; Albonesi, D. H. The energy impact of aggressive loop fusion. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques* (2004), IEEE Computer Society, pp. 153–164.