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 <XXX-MES> de <XXX-ANO>.

BANCA EXAMINADORA

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

Prof. <NOME DO AVALIADOR>, <INSTITUIÇÃO>

Prof. <NOME DO AVALIADOR>, <INSTITUIÇÃO>

Prof. <NOME DO AVALIADOR>, <INSTITUIÇÃO>

Prof. <NOME DO AVALIADOR>, <INSTITUIÇÃO>

Niterói

<ANO>

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 Tavares 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 Esteban Clua, Lúcia Drummond 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, até o momento, as decisões de hardware 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 e usa o algoritmo Guloso e Programação Dinâmica para resolver-los. 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 increases the power of these devices, since it makes possible the concurrent execution of different tasks on the same hardware. However, the hardware decisions about the order execution depends heavily the order at which the kernels are submitted, causing the utilization of the hardware is not optimized. This work propose 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 modeling as a knapsack problems and use algorithm greedy and dynamic programming approach to solve them. The results are measure using kernels with different sizes and resource requirements, resulting a significant gains in the average turnaround time and system throughput compared with a standard execution of concurrent kernels.

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

Lista de Figuras

Lista de Tabelas

Sumário

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.

O *Survey* [30] introduziu o termo *General-Purpose Computation* para discutir aplicações não gráficas que usufruem do poder computacional das GPUs para resolver problemas de desempenho de algoritmos.

Naquele momento, programar para essa GPU significava expressar os algoritmos da aplicação em operações sobre estruturas de dados gráficos (especialmente *pixels* e vetores) [30]. 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 [30] utilizando essas GPUS, as fabricantes começaram a alterar a arquitetura da GPUs para introduzir o conceito de *General Purpose Graphics Processing Unit* (GPGPU). Com essa mudança, as GPUs passaram a suportar um conjunto de instruções de propósito genérico e foram criadas ferramentas, *frameworks* e linguagem de programação (CUDA, OpenCL) para usufruir melhor desse potencial computacional.

Com a introdução de GPGPU, 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 a CPU como processador central, isto é, o controle de fluxo da aplicação da GPU requer que a CPU gerencie o fluxo de dados e a sincronização das tarefas [25][10][8]. A Figura 1.1 ilustra esse trabalho colaborativo.

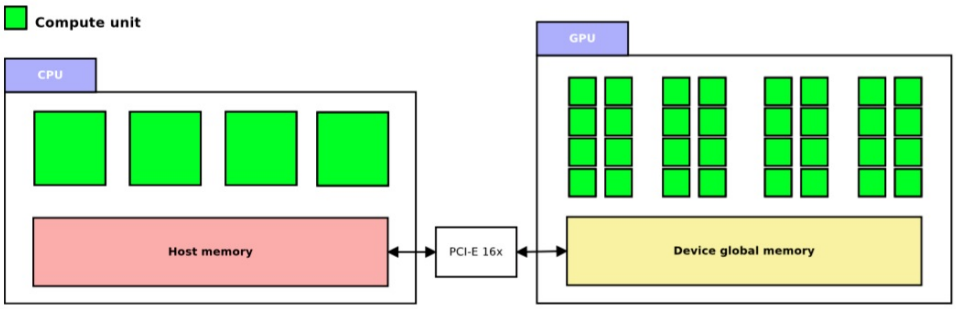

Figura 1.1: Arquitetura CPU e GPU [8]

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 as 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 [42][15][27]. A Figura 1.2 ilustra uma execução sequencial e concorrente de diferentes arquiteturas.

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][42]. Em seguida, com o lançamento da tecnologia Hyper-Q já na arquitetura Kepler [29] possibilitou-se que diferentes contextos de aplicações executassem algoritmos concorrentemente. A Figura 1.3 ilustra a limitação de uma única fila de trabalho na Fermi [7] onde um conjunto de kernels de mesma letra (A e B) devem ser executados concorrentemente entre si. A Figura 1.4 ilustra a execução na Kepler [7] com o mesmo conjunto de kernels, onde todos da mesma letra foram executados concorrentemente entre si.

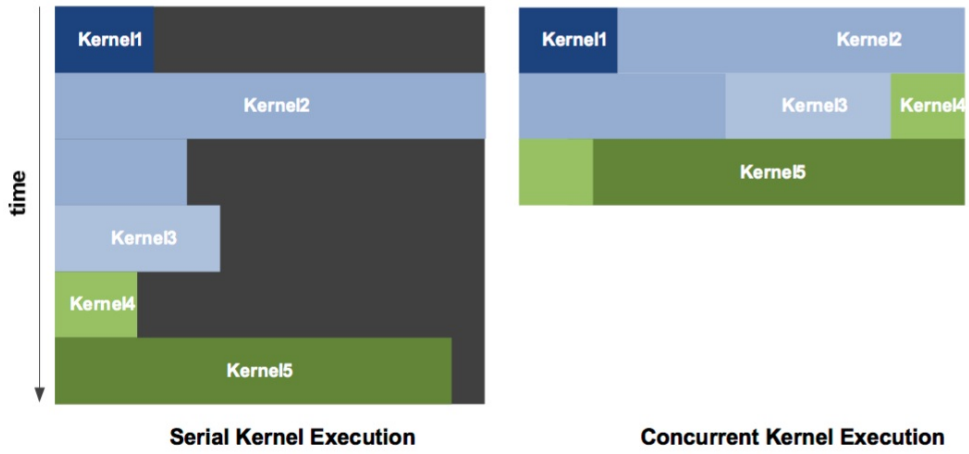

Figura 1.2: Concorrência de kernels na arquitetura Fermi [27]

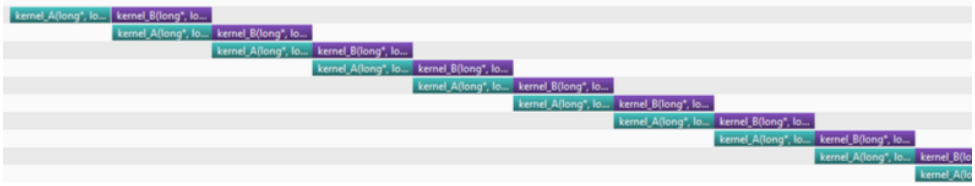

Figura 1.3: Execução concorrente entre kernels de mesma cor na arquitetura Fermi [7] onde o eixo x representa o tempo

Figura 1.4: Execução concorrente entre kernels de mesma cor na arquitetura Kepler [7] onde o eixo x representa o tempo

1.2 Arquitetura da GPU

As GPUs são co-processadores dedicados para processamento gráfico de 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 pelos dados.

Para facilitar a *interface* entre o programador e a GPU, foi criado a linguagem CUDA que fornece abstrações simples sobre a hierarquia de threads, memoria e sincronização. O modelo de programação em CUDA permite ao desenvolvedor escrever programas denominados de kernels sem a necessidade de conhecer previamente detalhes e do tipo de

hardware que será executado. A arquitetura CUDA suporta diversas linguagens de programação, incluindo C, Fortran e OpenCL. Esse ambiente tem sido amplamente utilizado por aplicações e trabalhos de pesquisa encontrado em *notebook*, computadores e *clusters* de computadores [45].

Uma diferença essencial entre CPU e GPU, é que a primeira dedica a maior quantidade do seu circuito ao controle e a segunda foca mais na ALU (*Arithmetic Logical Units*), o que a torna muito mais eficaz para cálculos matemáticos [12][45]. A Figura 1.5 ilustra a diferença do circuito da CPU e da GPU.

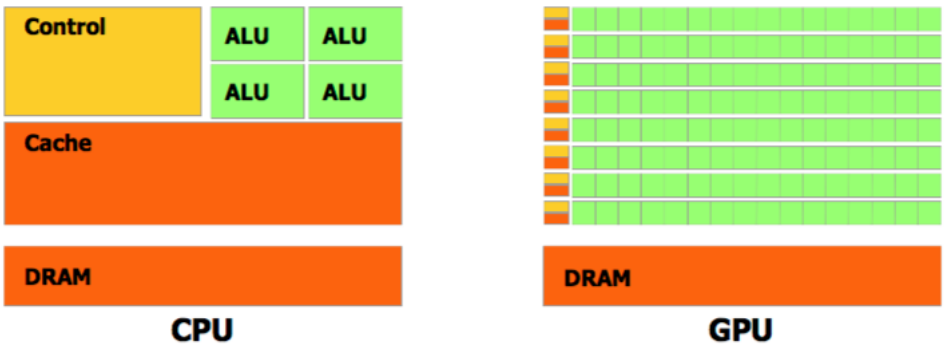

Figura 1.5: Diferença do circuito da CPU e GPU

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

Uma GPU Kepler GK110 [29], por exemplo, possui 15 SMs [11]. Cada geração de GPUS implementa variações na arquitetura dos seus SMs, sendo comum dar-lhes nomes diferentes. Na arquitetura Kepler, por exemplo, a SM é denominada de SMX. Todas as SMs possuem acesso a uma memória global que pode ser compartilhada por todos os SMs e também pode ser escrita pela CPU quando se deseja transferir dados de entrada para a execução de uma kernel. A Figura 1.6 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 texture onde os dados são para leitura [29]. A Figura 1.7 mostra a arquitetura de uma SMX da Kepler GK110.

A arquitetura Fermi foi pioneira no conceito de concorrência na execução de kernels,

Figura 1.6: Arquitetura de uma Kepler GK110 com 15 SMX

Figura 1.7: Arquitetura de uma SMX da Kepler GK110 com 192 CUDA *cores*

mas foi na arquitetura Kepler que a concorrência ocorreu com maior intensidade em função da tecnologia Hyper-Q. Essa tecnologia permitiu que ocorresse até 32 filas de *hardware* de execuções simultâneas, ao passo na Fermi apenas uma conexão poderia ocorrer [29]. A Figura 1.8 ilustra a diferença entre as filas.

As filas de execução concorrentes são gerenciadas pelos Streams. Um Stream é uma

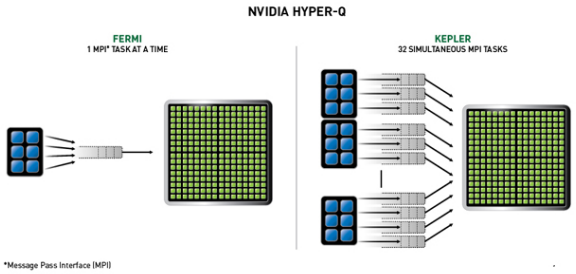

Figura 1.8: A concorrência da Fermi é limitada a uma única fila de trabalho. Na Kepler, as *streams* podem executar concorrentemente usando filas separadas de tarefas

fila a nível de software 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 [31]. Na arquitetura Kepler, a GPU possui somente 32 filas de *hardware*, não tendo limite bem definido de filas de *software* (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* [31][29].

A Figura 1.9 mostra um modelo de execução de uma mesma submissão para GPU em diferentes arquiteturas. Na Fermi, apenas as tarefas (C,P) e (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 [29].

A Figura 1.9 mostra um modelo de execução de uma mesma submissão para GPU em diferentes arquiteturas. No modelo da Fermi, apenas as tarefas (C,P) e (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 [29].

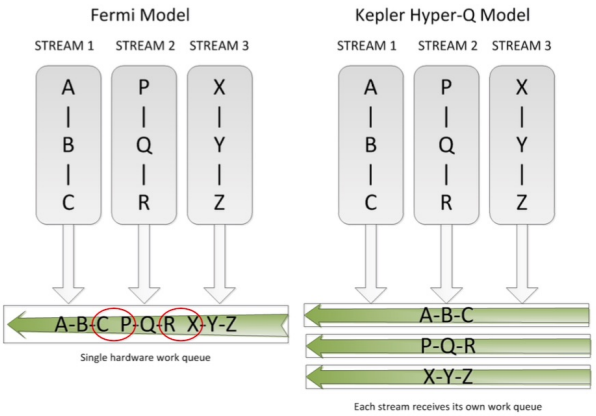

Figura 1.9: Diferenças de concorrência entre a Fermi e a Kepler

A tecnologia Hyper-Q oferece um benefício significante no uso de sistemas compu-

tacionais paralelos baseado em MPI (*Message Passing Interface*). A utilização de MPI para fazer acesso as GPUs pode trazer compartilhamento do recurso. Porém, na 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 dependência aumentando dramaticamente a eficiência do compartilhamento da GPU nos processos do MPI [19].

Após a arquitetura Kepler, a Maxwell foi lançada com um aumento de 2 vezes o desempenho por *watts* [11]. 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.

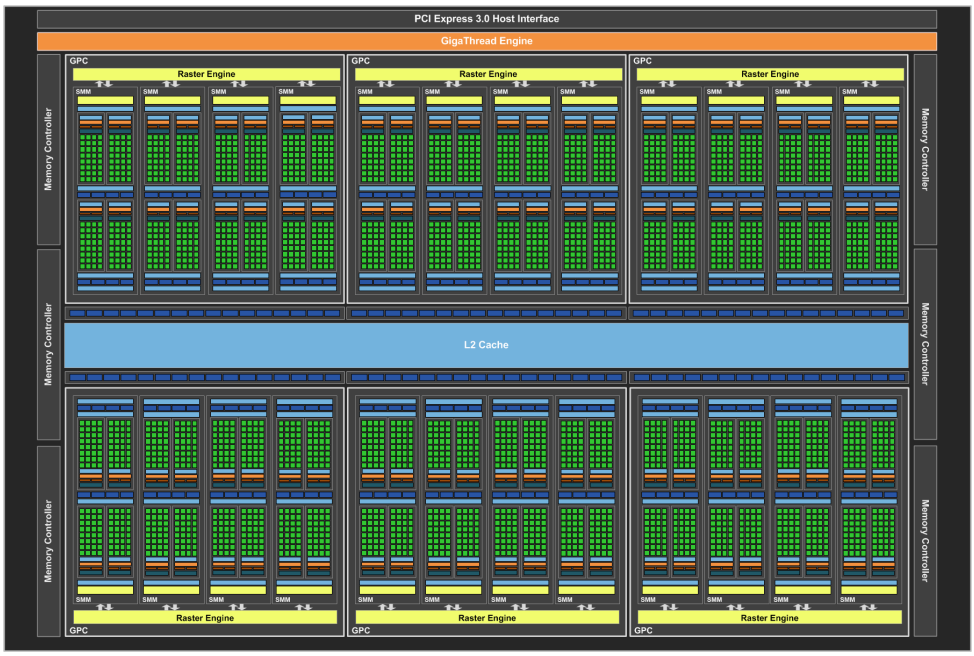

Figura 1.10: Arquitetura da Maxwell com 6 grupos de GPC

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 1.11. A figura 1.11 mostra um kernel sendo particionado em bloco para serem executados independentemente. Assim, GPUs com mais SMs podem automaticamente executar o programa com menor tempo do que GPUs com poucos SMs.

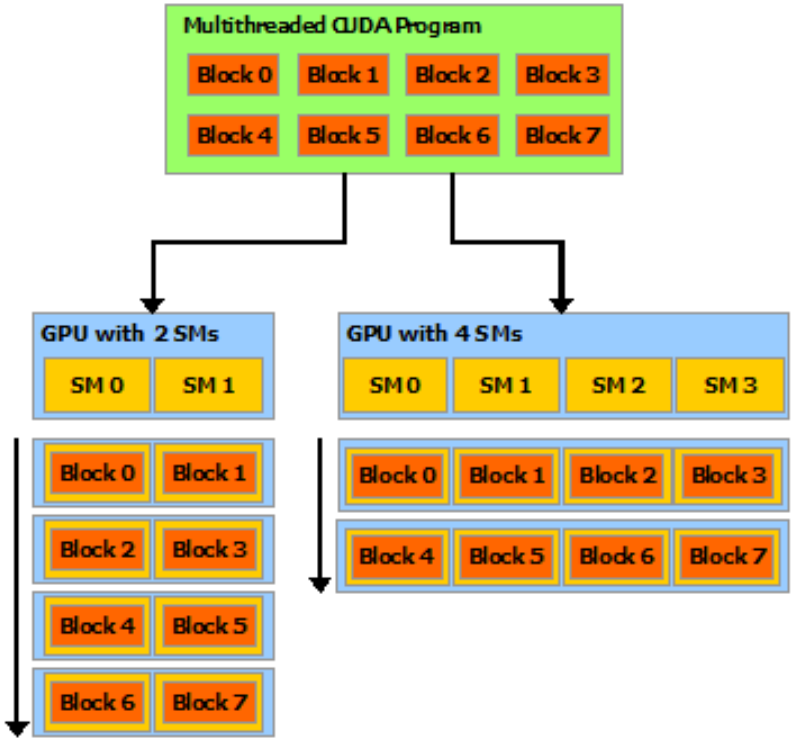

Figura 1.11: Particionamento de um kernel para executar em qualquer arquitetura

1.3 Escalonamento das tarefas

A política de escalonamento dos kernels é proprietário da NVIDIA e nenhuma informação está documentada. Existem entretanto especulações que a politica de escalonamento segue a estratégia *left-over*[31][46][34]. O escalonador dispara os blocos do kernel na primeira fila de *hardware*, se houverem recursos suficientes. Os próximos disparos são feitos nas próximas filas de *hardware* seguinte 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 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 1.12, 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

A Figura 1.12 mostra 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.

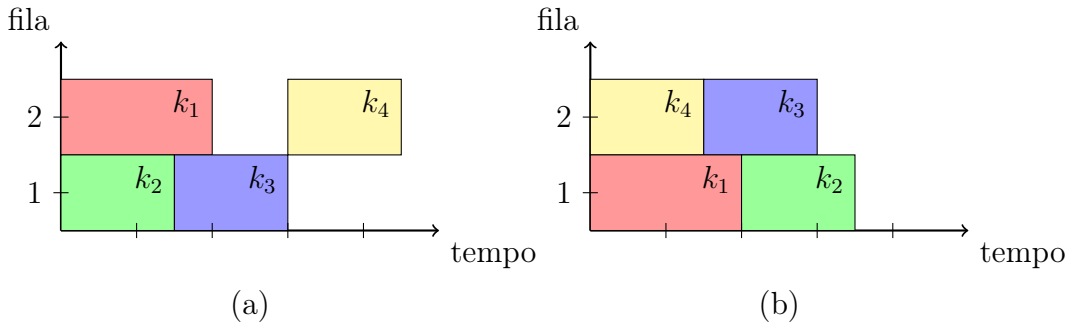

Figura 1.12: Diferente ordem de execução de kernels gerando diferentes taxa de ocupação

O presente trabalho irá propor uma estratégia de ordenação para submissão de múltiplos kernels, com o objetivo de melhorar os *turning around time* dos kernels. A proposta da ordenação é escolher um conjunto de kernels que combinados utilizam melhor dos recursos da GPU de forma concorrente. Para isso, utilizamos o problema da mochila [2][35] para efetuar a seleção, sendo que o resultado representa a maximização do uso dos recursos dos kernels. A capacidade da mochila foi associado a 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 foi modelado como sendo os recursos de cada kernel e o valores de cada item da mochila foi mapeado para 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 problema da mochila foi implementado através de programação dinâmica [23][3][4], que maximiza o somatório dos valores acumulados.

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

Capítulo 2

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: fusão de kernels, Divisão de kernels, Contextos de aplicação e Ordenação de kernels.

2.1 União de kernels

Nas primeiras tentativas de simulação de kernels concorrentes, alguns trabalhos foram 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. [33] consiste em executar um kernel persistente onde cada bloco de threads observa um espaço de memória dedicado esperando, em um laço infinito, 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. [41] 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. [47].

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

2.2 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. [46] 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. [31] 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 solicitado 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.

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.

2.3 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. [42] explora a execução de kernels de diferentes contextos de aplicação através de um contexto compartilhados chamado *context funneling*. Através dessa abordagem, esse contexto compartilhado é usado como serviço para executar kernels de diferentes contextos de aplicação.

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.

O Ravi et al. [34] 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.

2.4 Ordenação de kernels

Finalmente, os trabalho que propõem alcançar o máximo de utilização com base na ordem em que kernels são invocados para GPU, chamado de reordenamento kernel, são os que mais se aproximam com a atual proposta. Wende et al. [44] 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] propõe um reordenação para a novas arquiteturas de GPU com tecnologia Hyper-Q. Sua abordagem calcula uma pontuação simbiótica, que 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 simbióticas 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.

2.5 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 2.1 apresenta diferentes linhas de solução em diferentes arquiteturas.

Embora melhorar a execução concorrente seja o objetivo de todos os trabalhos, muitos perderam sua importância prática, em função de considerarem que o hardware não poderia conter concorrência explícita.

Trabalho	Classificação	Arquitetura	Teste
Guevara et al., 2009 [16]	União	Tesla	10 Parboil benchmark
Wang et al., 2010 [41]	União	Tesla	CUDA SDK SPEC2K benchmark
Peters et al., 2010 [33]	União	Tesla	MatMul e Sorting
Ravi et al., 2011 [34]	União	Fermi	Image Processing PDE Solver, K-Means Binomial, BlackScholes K-Nearest Neighbours Molecular Dynamics
Wang et al., 2011 [42]	União	Fermi	Scalable Synthetic Compact Application
Gregg et al., 2012 [15]	União	Fermi	Bitonic Sort, Kmeans Floyd Warshall Nearest Neighbor Simple Convolution Fast Walsh Transform
Awatramani et al., 2013 [5]	Divisão	Fermi	Rodinia benchmark
Pai et al., 2013 [31]	Divisão	Fermi	Parboil2 benchmark
Zhong et al., 2014 [46]	Divisão	Fermi	CUDA SDK Parboil Benchmark CUSP
Jiao et al., 2015 [18]	Divisão	Kepler	CUDA SDK Rodinia Benchmark
Tanasic et al., 2014 [37]	Contexto	Kepler	Parboil Benchmark
Park et al., 2015 [32]	Contexto	Kepler	Rodinia Benchmark Parboil Benchmark
Adriaens et al., 2012 [1]	Ordenação	Tesla	Microbenchmarks Parboil benchmark
Wende et al., 2012 [44]	Ordenação	Fermi	Molecular Dynamics Simulation
Liang et al., 2015 [21]	Ordenação	Kepler	CUDA SDK
Li et al., 2015 [20]	Ordenação	Kepler	BlackScholes SmithWaterman Electrostatics Embarrassingly Parallel

Tabela 2.1: Trabalhos relacionados de escalonamento da GPU

Capítulo 3

Algoritmo de Ordenação

3.1 Definição do Problema

Diferentes kernels podem ser executados em qualquer ordem concorrentemente desde que a GPU tenha recursos suficiente para executar. Quando um kernel não conseguir executar concorrentemente, por não haver recursos disponível, 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 *turnaround time*. A Figura 1.12 e a Tabela 1.1 mostram duas diferentes ordens de execução dos kernels gerando diferentes taxa de ocupação da GPU.

Este trabalho tem como objetivo ordenar os kernels antes de serem submetidos para GPU de tal forma a melhorar o *turnaround time* dos kernels. Para tal, dado 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 NSM e que os recursos de cada SM é $R_j \mid j \in \{1, 2, 3\}$, 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

O trabalho utiliza de uma heurística de calculo de capacidade, onde para um dispositivo D a capacidade total de um recurso W_j representa o somatório das capacidades de

cada SM. Por tanto, a equação 3.1 mostra o valor da capacidade total do dispositivo D.

$$W_j = R_j \times NSM \mid j \in \{1, 2, 3\} \tag{3.1}$$

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

- w_{i1}: a quantidade de memória compartilhada por bloco
- w_{i2}: o número de registradores por bloco
- w_{i3}: o número de threads por bloco
- t_i^{est}: tempo estimado de execução

A quantidade de memória compartilhada, número de registradores e número de threads podem ser obtidos através de ferramenta de *Profiler* da NVIDIA[28]. O tempo estimado da execução de um kernel pode ser retirado de uma execução anterior ou de técnicas de estimativa de tempo[22].

3.2 Processo de ordenação

Para que o conjunto de N kernels submetidos para a GPU tenha um bom *turnaround time*, 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*.

A Figura 3.1 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 3.1, 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 3.2 mostra as taxas de ocupação $TO_{\alpha\beta} \mid \alpha \in \{a, b\}, \beta \in \{0, ..., N\}$ descrita na equação 3.2 das ordenações Ord_α para cada instante T_β e o cálculo da média ponderada WAT (*Weighted Average Time*) ilustrada na Figura 3.2 descrito na

equação 3.3. A Figura 3.2 também mostra que a Ord_b tem um WAT maior e que o tempo de execução total foi reduzido.

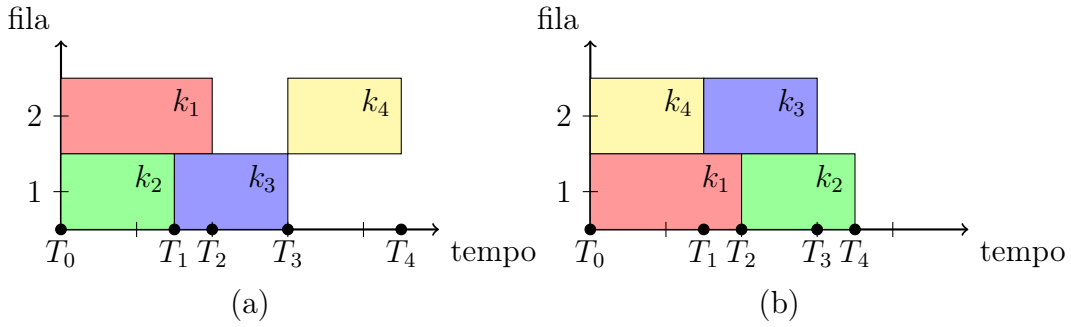

Figura 3.1: $N = 4$ Kernels com diferente instantes $T_\beta \mid \beta \in \{0, 1, 2, 3, 4\}$ na simulação do calculo da ordem de execução.

	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 3.1: Valores de x_i para cada instante T_i

$$TO_{\alpha\beta} = \sum_{i=1}^{N} w_i x_{\alpha i} \times (T_{\beta+1} - T_\beta) \tag{3.2}$$

$$WAT_\alpha = \sum_{\beta=0}^{N} TO_{\alpha\beta} \div \sum_{i=1}^{N} t_i^{est} \tag{3.3}$$

$TO_{\alpha\beta}$	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 3.2: Taxa de ocupação para cada instante T_i

O processo de ordenação de kernels consistem na simulação da execução do escalonamento de todos eles. Para que a simulação possa ser construída é preciso primeiro criar um grupo de kernels que comporão a lista de execução e a cada termino 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 termino de um e a entrada de outros kernels correspondem à ordem com que os kernels devem ser executados.

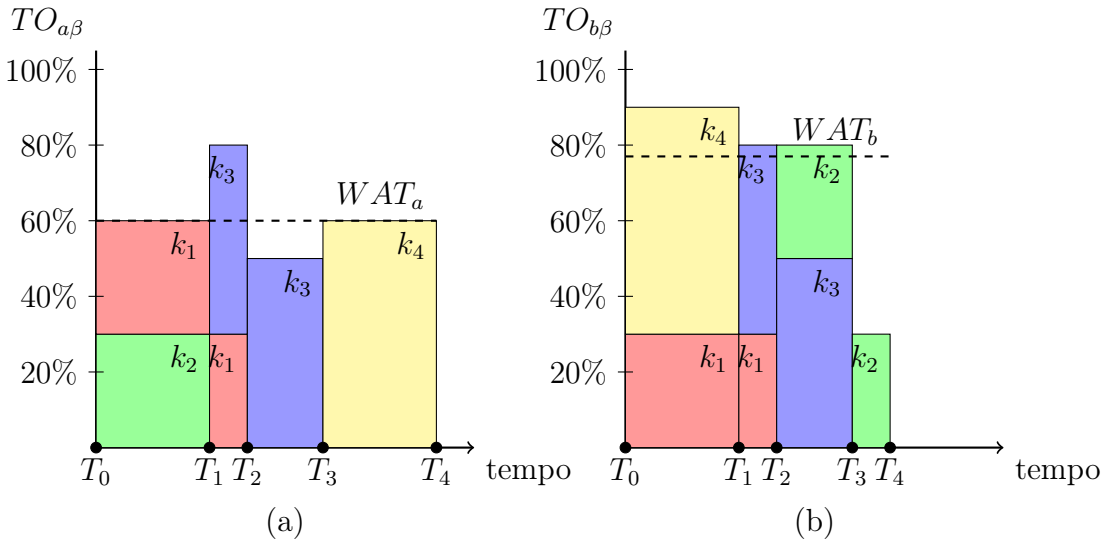

Figura 3.2: Taxa de ocupação $TO_{\alpha\beta}$ dos exemplos de ordenação Ord_α para cada instante T_β e também apresenta uma linha tracejada com a média ponderada da $TO_{\alpha\beta}$

No inicio da execução da ordenação T_0, assume-se que todos os recursos do dispositivos D estão disponíveis. Neste momento, será selecionado um conjunto de kernels para iniciar a simulação da execução.

Para um certo instante $T_i \mid i \in \{0, ..., i, ...N\}$ da execução da ordenação, um conjunto de kernels já foram 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, incrementa o dispositivo $W_j^{av} \mid j \in \{1, 2, 3\}$ e busca-se por outros kernels que ainda não foram submetidos.

A Figura 3.3 ilustra o instante T_i para os dois exemplos de ordenação Ord_α. Na parte (a), o kernel k_2 libera os recursos para o dispositivo D através do 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 para o kernel k_3 entrar em execução.

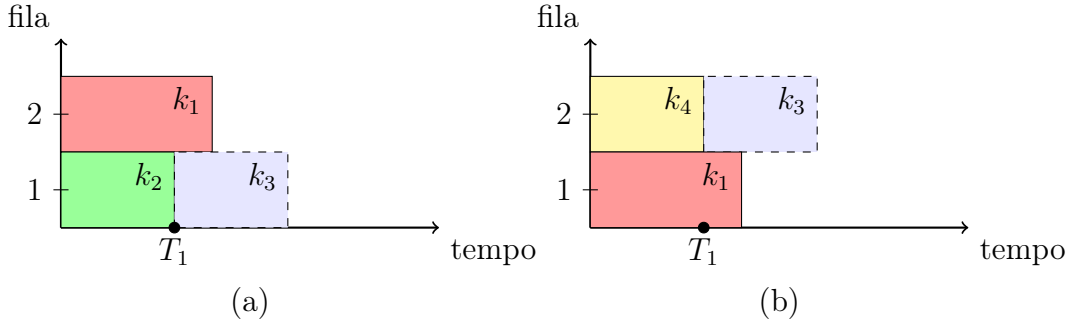

Figura 3.3: Instante T_i dos dois examples de ordenação

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 3.4 mostra o resultado da lista criada a partir do *Round-Robin* das filas construídas.

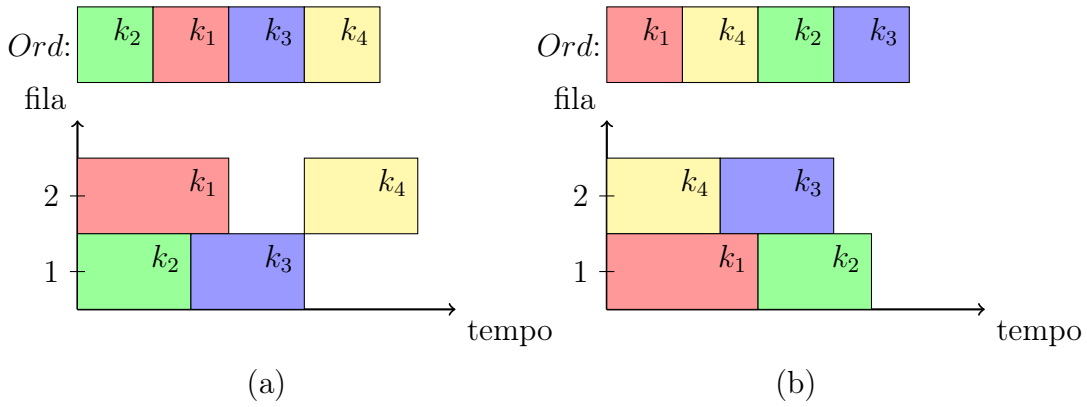

Figura 3.4: Instante T_N e a construção da lista utilizando o modo *Round-Robin*

Neste trabalho, a escolha da submissão simulada dos novos kernels é feita através da solução do problema da mochila multidimensional, onde os kernels representam os itens da mochila e a capacidade da mochila representa os recursos disponível W_j^{av} do dispositivo. O problema da mochila precisa ser multidimensional porque um kernel possui diferentes tipos de recursos independentes, onde todos elas devem estar disponíveis para que o item entre na mochila.

3.3 Problema da Mochila 0-1 Multidimensional

O Problema da Mochila Multidimensional (PMM) [14][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.

Normalmente este problema é resolvido com Programação Dinâmica, obtendo a resolução exata do problema, mas também pode ser resolvido com o Algoritmo Guloso e Meta-Heurística (algoritmos genéticos) para soluções aproximadas.

Suponha-se N itens distintos de M dimensões e uma mochila com capacidade $W_j \mid j \in \{1, ..., M\}$. Seja $x_i \mid i \in \{1, ..., N\}$ os itens que podem ser carregado quando $x_i = 1$ e $x_i = 0$ quando o item não pode ser carregado. 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 3.4

$$sujeito\ a \sum_{i=1}^{N} w_{ij}x_i \leq W_j \mid j \in \{1, ..., M\} \tag{3.4}$$

Cada item da mochila possui um valor $v_i \mid i \in \{1, ..., N\}$. Achar o valor máximo da mochila é maximizar a equação 3.5:

$$maximizar \sum_{i=1}^{N} v_i x_i \tag{3.5}$$

3.4 Adaptação do problema para GPU

O problema da mochila está sendo utilizado para selecionar kernels que irão ser processados concorrentemente a cada instante T_i. 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 3.6 onde a mochila está sujeita para cada dimensão foi modificada para $M = 3$.

$$sujeito\ a \sum_{i=1}^{N} w_{ij}x_i \leq W_j \mid j \in \{1, 2, 3\} \tag{3.6}$$

A solução para o problema da mochila implementado com Programação Dinâmica dispõem de uma solução ótima [24][23][6][38][3] para o somatório dos valores dos itens da mochila. Mapeando os valores dos itens da mochila expresso pela equação 3.7 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.

$$v_i = \frac{\frac{w_{i1}}{W_1} + \frac{w_{i2}}{W_2} + \frac{w_{i3}}{W_3}}{3 \times t_i^{est}} \tag{3.7}$$

Para uma GPU com arquitetura Kepler, o número de filas de *hardware* é 32 ($NQ = 32$). Assim, o algoritmo irá utilizar um conjunto de filas Q para ordenar os kernels k_i nas filas. O algoritmo de ordenação consiste em escalonar os kernels k_i nas filas usando o modo *Round-Robin* enquanto tiver recursos disponíveis.

3.5 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.

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 passe a trabalhar como descrito na Equação 3.8 e os recursos w_{ij} necessários para o kernel k_i executar será mapeado para Equação 3.9.

$$W = W_1 \times W_2 \times W_3 \tag{3.8}$$

$$w_i = (w_{i1} \times W_2 \times W_3) + (w_{i2} \times W_2) + w_{i3} \tag{3.9}$$

3.6 Algoritmo de Ordenação

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. Este consiste em um *loop* (linha 2) enquanto todos os kernels não foram submetidos, onde cada instante T_i é um iteração desse *loop*. No instante T_i, 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 o 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_i, 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_i, as filas estarão preenchidas com os kernels selecionados e serão retornados os kernels na ordem das filas seguindo o modo *Round-Robin*.

A implementação dos procedimentos $FindShortQueue$ não serão detalhados nesse trabalho porque ele pode ser resolvida através de uma fila de prioridade [39][40] (*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

Algorithm 1 Algoritmo de ordenação dos Kernels

```
1: function KernelReorder(KernelList, W^av)
2:     while KernelList not empty do
3:         NextSubmitList ← GetKernelsToSubmit(KernelList, W^av)
4:         for each k_i ∈ NextSubmitList ∧ i ≤ NQ do
5:             q_j ← FindShortQueue(Q)
6:             q_j ← q_j + k_i
7:             W^av = W^av − w_i
8:             KernelList = KernelList − k_i
9:         q_j ← FindShortQueue(Q)
10:        k_e ← q_j.last
11:        W^av = W^av + w_e
12:    return RoundRobingCross(Q)
```

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_i. Esse procedimento irá implementar o problema da mochila com as adaptações descritas nesta sessão utilizando o algoritmo de Programação Dinâmica e Guloso.

3.6.1 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 3.10 mostra a equação

que denota o valor máximo obtido na mochila ao combinar os valores dos pesos w_i com a inclusão dos kernels x_i.

$$\begin{aligned} OPT(i,w) &= \max\left(\sum_{i=1}^{N} v_i x_i\right) \\ sujeito\ a &= \sum_{i=1}^{N} w_{ij} x_i \leq W_j \mid j \in \{1,2,3\} \end{aligned} \tag{3.10}$$

A partir da Equação 3.10, o cálculo de recorrência para o valor máximo obtido na mochila pode ser expressa pela Equação 3.11.

$$\begin{aligned} OPT(i,w) &= 0, i = 0 \\ &= OPT(i-1,w), w_i > w \\ &= \max(OPT(i-1,w), v_i + OPT(i-1, w-w_i)) \end{aligned} \tag{3.11}$$

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 $G(i-1,w)$ e o possível novo estado $G(i-1,w-w_i)+v_i$ onde é colocado o item i, reduzido a capacidade da mochila w_i e incrementado o valor v_i.

Com a equação de recorrência 3.11, iremos usar uma matriz M para armazenar e consultar os resultados dos valores $OPT(i,w)$ anteriormente processados. O Algoritmo 3 implementa a criação da matriz M utilizando esta equação de recorrência.

Algorithm 3 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 $GetKernelsToSubmit$ implementando no Algoritmo 4 usará a matriz M para recuperar os kernels k_i selecionados e retorná-los

na ordem decrescente de valor.

O Algoritmo 4 inicia o conjunto dos kernels selecionados como vazio (linha 2). Ao longa da execução, a quantidade de memória disponível será decrementada toda vez que se selecionar um kernel k_i (linha 8). Para isso, foi criado uma variável auxilia $tempW$ que indica a quantidade de memória disponível no instante da 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 decrementado os recursos disponível pelo w_i (linha 8 e será incluído o k_i no conjunto de kernels selecionados. Por fim, ao terminar toda a iteração pelos N kernels, a lista $SelectedKernels$ será ordenada na ordem decrescente pelo valor para que quando houver um corte de NQ kernels, os de maior valor sejam selecionados.

Algorithm 4 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
```

3.6.2 Algoritmo Guloso

O método Guloso é uma técnica mais simples que pode ser aplicada a uma grande variedade de problemas. A grande maioria destes problemas possui N entradas e é requerido obter um subconjunto que satisfaça alguma restrição. Qualquer subconjunto que satisfaça esta restrição é chamado de solução viável. Deseja-se então encontrar uma solução viável que maximize ou minimize uma dada função objetivo.

A implementação do Problema da Mochila utilizando Guloso é bem mais simples do que a Programação Dinâmica. O Algoritmo 5 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 6), se houverem recursos suficiente $tempW$ para k_i (linha 7), os recursos w_i do kernel serão decrementados (linha 8) e ele será selecionado (linha 9).

Algorithm 5 Implementação do $GetKernelsToSubmit$ usando o Guloso como implementação do problema da mochila

```
1: function GetKernelsToSubmit(KernelList, W^av)
2:     SelectedKernels ← ∅
3:     tempW ← W^av
4:     SortedKernelList ← SortDecrementValuesPerWeight(KernelList)
5:     for each k_i ∈ SortedKernelList do
6:         if w_i < tempW then
7:             tempW ← tempW − w_i
8:             SelectedKernels ← SelectedKernels ∪ k_i
9:     NextSubmitList ← OrderSet(SelectedKernels)
10:    return NextSubmitList
```

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.

3.6.3 Consumo de Memória

A implementação do problema da mochila usando Programação Dinâmica ou Guloso apresentam vantagens e desvantagens para cada método, onde as duas abordagem trazem soluções boas.

A Programação Dinâmica é muito utilizado quando a diferença é impactante para a aplicação entre uma solução boa e ótima. 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 posto na mochila com a capacidade dela. Dependendo da quantidade de itens e da capacidade da mochila, o produto cartesiano deles pode tornar o método inviável.

Em função da simplicidade do método Guloso, o grau de processamento e de espaço não costuma ser um problema. Porém, a heurística utilizada de ordenação dos itens em cima da razão entre o valor e o peso representa uma boa solução num contexto local, não dando a melhor resultado para o problema.

Capítulo 4

Resultados

4.1 Ambiente Computacional

Antes de apresentar os resultados obtidos neste trabalho, a tabela 4.1 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	24	15
Shared Memory por SM	96KB	48KB
Numero of Registers por SM	64K	64K
Max número de threads por SM	2048	2048
Arquitetura	Maxwell	Kepler
Pico de desempenho	6.6 TFLOPs	4.2 TFLOPS

Tabela 4.1: Configuração das GPUs nos resultados

4.2 Metodologia

A execução de kernels concorrentes é uma funcionalidade disponível em GPUs da NVIDIA com arquiteturas Kepler em diante. 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 foi construído pensando em cenários onde kernels gerenciados pela GPU são executados de forma concorrente, podendo ou não ser de uma mesma aplicação.

Existem diversos *benchmarks* focados em CUDA na literatura, tais como Rodinia [9], Parboil [36] e CUDA SDK [26], composto por um conjunto 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 inteiro alocado na memória compartilhada. Uma quantidade de registradores também são alocados, 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 ordenação. 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 distintos de tamanho de kernel, onde N varia de $\{32, 64, 128, 256\}$. 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 varia de $\{32, 64, 128, 256, 512\}$. Todos os números atribuído no experimento vem de um gerador aleatório. Para cada experimento com N kernels, foi gerado 50 diferentes conjuntos de kernels com valores aleatórios de blocos e recursos para minimizar o efeito da aleatoriedade dos testes.

Neste trabalho, foram criados diversos grupos de kernels, cada um com N kernels

diferentes, onde N varia de $\{32, 64, 128, 256\}$. Em cada conjunto, foi também variado o número máximo e mínimo de blocos a serem instanciados para cada chamada de kernel. A quantidade mínima de blocos é sempre 4 e a quantidade máxima NB é limitado a um valor limitado por $\{32, 64, 128, 256, 512\}$. Todos os números atribuídos no experimento vem de um gerador aleatório. Para cada experimento, composto por 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 $ANTT$, proposto por Eyerman [13][43], é calculada pela equação 4.1 e representa a média normalizada da razão entre o TT_i e o TT_i^{Alone} de todos os kernels para uma ordenação, sendo que TT_i^{Alone} representa o valor do *turnaround time* da execução do k_i quando o kernel é executado sozinho. 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.

$$ANTT = \frac{1}{N} \sum_{i=1}^{N} \frac{TT_i}{TT_i^{Alone}} \tag{4.1}$$

A métrica STP, também proposto por Eyerman [13][43], é calculada pela equação 4.2 e representa o somatório da fração dos TT_i^{Alone} com os TT_i para cada ordenação. Em outras palavras, esta métrica mostra a quantificação acumulada de um programa sequencial perante um concorrente. Nesta métrica o valor deve estar no intervalo de $\{1, ..., N\}$ e o quanto maior, melhor.

$$STP = \sum_{i=1}^{N} \frac{TT_i^{Alone}}{TT_i} \tag{4.2}$$

4.3 Desempenho

Este trabalho implementou duas técnicas de ordenação de kernels utilizando o Problema da Mochila com o método Guloso (*Greedy*) e Programação Dinâmica (*Dynamic*). O

resultado da ordenação desses dois métodos serão comparados com a ordenação do programa (*Standard*), onde os N kernels são ordenados na ordem crescente pelo índice. Por exemplo, para $N = 4$, a ordenação *Standard* corresponderá a $Ord = \{k_1, k_2, k_3, k_4\}$.

As figuras 4.1 e 4.3 mostram o *profile* da execução dos kernels ordenados 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. Em ambos os testes, a GPU suporta até 32 filas de trabalhos simultâneos onde o *trace* da ordenação utilizou de até 13 filas para o *Standard*, 17 para *Greedy* e 20 para *Dynamic*.

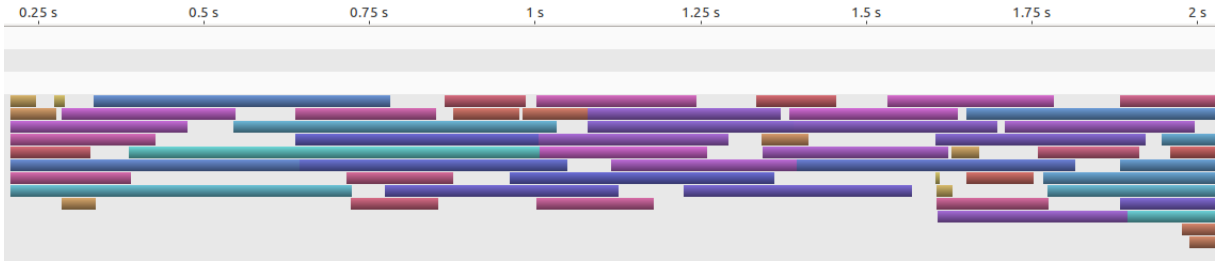

Figura 4.1: *Trace* de execução da ordenação *Standard* com 64 kernels

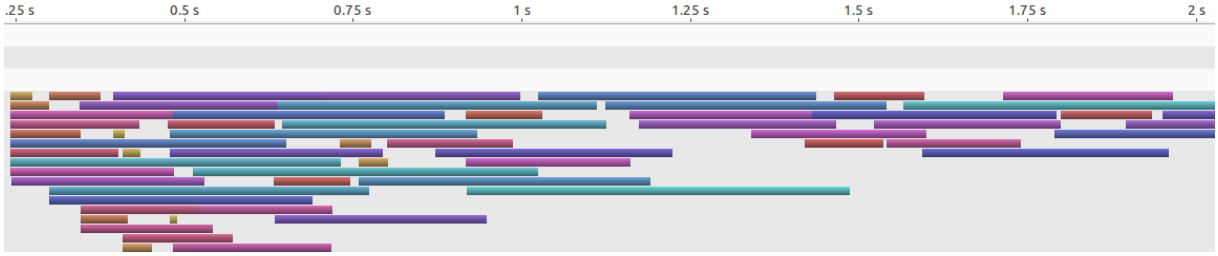

Figura 4.2: *Trace* de execução da ordenação *Greedy* com 64 kernels

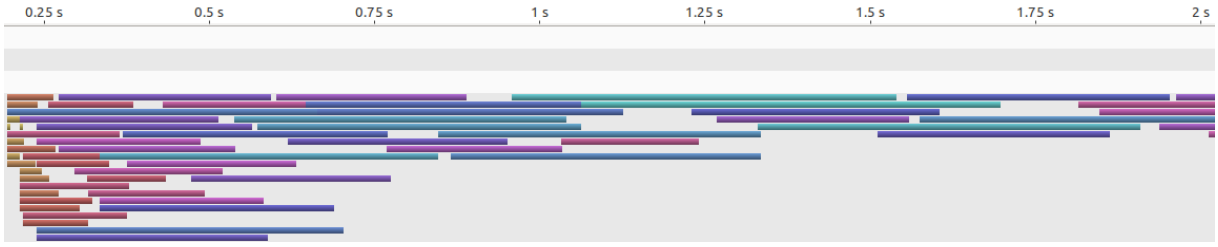

Figura 4.3: *Trace* de execução da ordenação *Dynamic* com 64 kernels

4.3.1 Average Normalized Turnaround Time

As Figuras 4.4 e 4.5 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* perante o *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 ordenação *Standard* com o *Dynamic* é significativo.

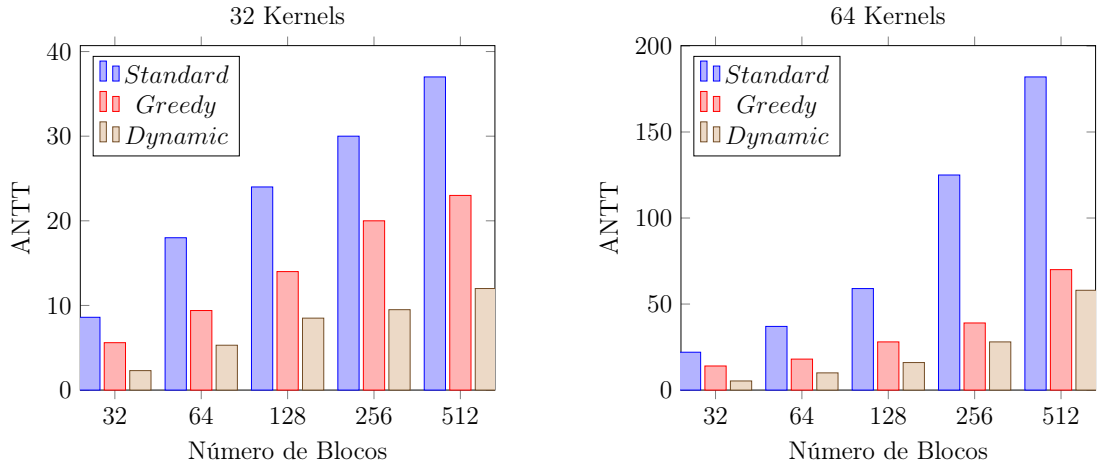

Figura 4.4: Resultado do ANTT para $N = 32$ e $N = 64$ na TitanX.

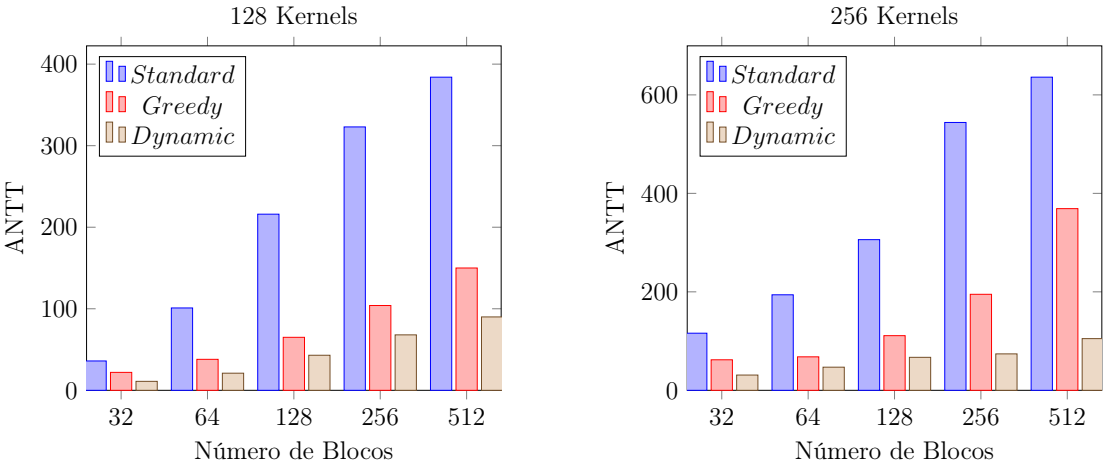

Figura 4.5: Resultado do ANTT para $N = 128$ e $N = 256$ na TitanX.

4.3.2 System Throughput

As Figuras 4.6 e 4.6 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 ordenação *Standard* com o *Dynamic* também é significativo.

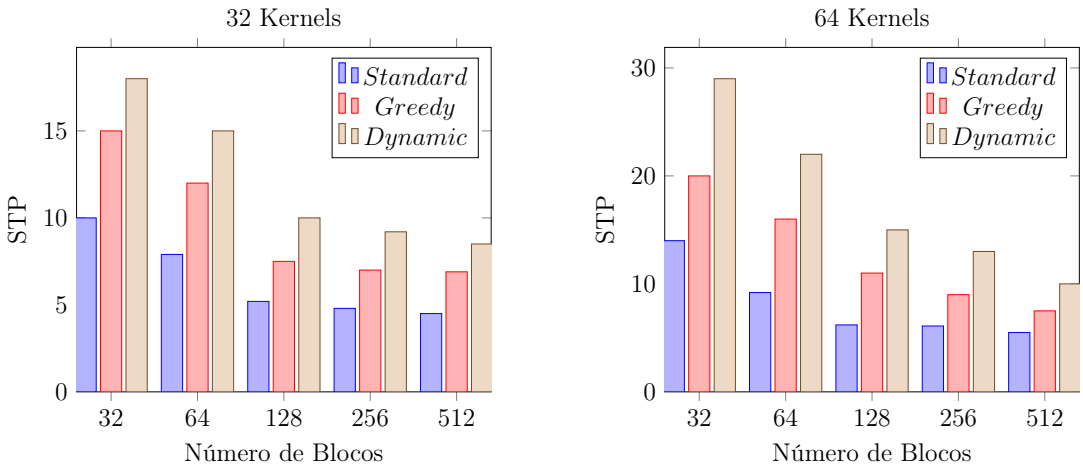

Figura 4.6: Resultado do STP para $N = 32$ e $N = 64$ na TitanX.

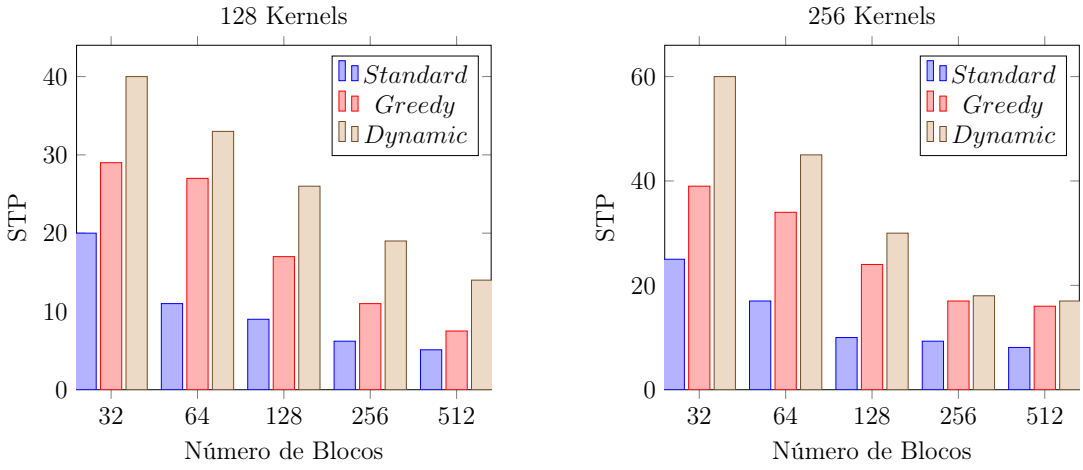

Figura 4.7: Resultado do STP para $N = 128$ e $N = 256$ na TitanX.

4.4 Overhead

O algoritmo de ordenação é realizado estaticamente antes de executar os kernels. Cada ordenação 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 ordenação *Greedy* para o *Dynamic* é significativo. Porém, o *overhead* de uma ordenação representa a razão do tempo de processamento com a diferença do *Average Turnaround Time* (ATT) entre a ordenação e o *Standard*. A equação 4.3 mostra a formula do ATT e a equação 4.4 mostra a fórmula do *overhead* O.

$$ATT_{\alpha} = \frac{\sum_{i=1}^{N} TT_i}{N} \mid \alpha \in \{greedy, dynamic\} \tag{4.3}$$

$$O_{\alpha} = \frac{OT_{\alpha} \times ANTT_{standard}}{(ANTT_{standard} - ANTT_{\alpha})} \mid \alpha \in \{greedy, dynamic\} \tag{4.4}$$

As tabelas 4.2 e 4.3 mostram os dois tipos de ordenação 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 ordenação fique pior. Porém, para que o *overhead* se preserve, a proporção irá se manter, mas o valor do ganho entre o $ATT_{dynamic}$ e $ATT_{standard}$ deverá aumentar para compensar o aumento do tempo de processamento. Neste trabalho foi utilizado a técnica de fator de escala, de forma a criar matrizes com $W \div 32$ colunas para cada unidade de recurso que é incrementado ao invés de criar uma matriz com W colunas.

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 4.2: *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 4.3: *Overhead* $O_{dynamic}$ da Programaçã Dinâmica na TitanX

4.5 Portabilidade

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.

4.5.1 Average Normalized Turnaround Time

As Figuras 4.8 e 4.8 mostram os resultados dos valores do $ANTT$ para uma $K40$ (Kepler). Pode ser observado pelas figuras que o padrão de comportamento da arquitetura Kepler se mantém semelhante ao Maxwell.

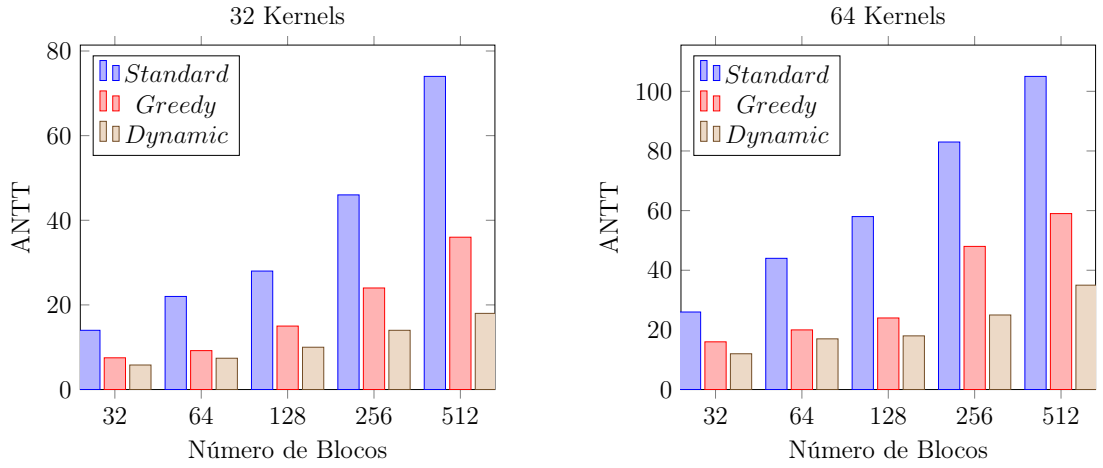

Figura 4.8: Resultado da K40 para $N = 32$ e $N = 64$ na K40 (Kepler).

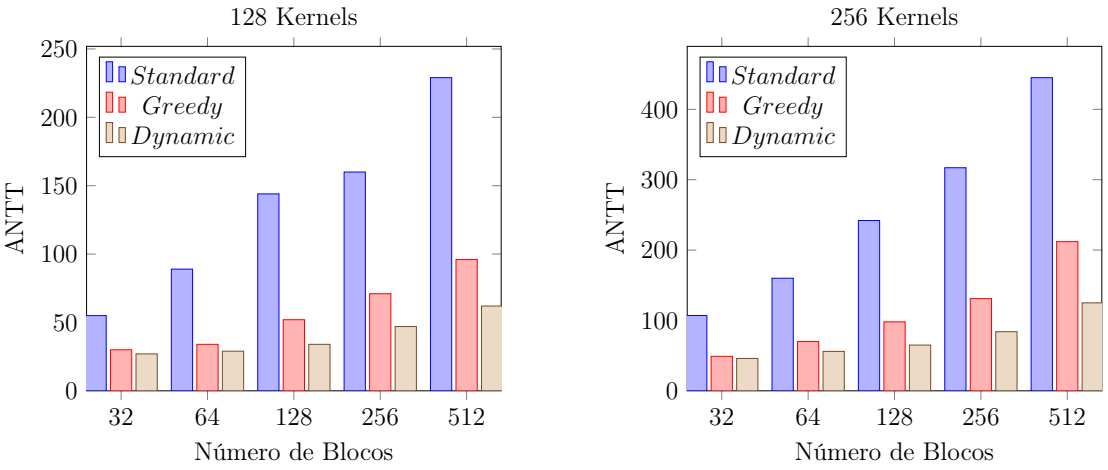

Figura 4.9: Resultado da K40 para $N = 128$ e $N = 256$ na K40 (Kepler).

4.5.2 System Throughput

As Figuras 4.10 e 4.10 mostram os valores do $ANTT$ para uma $K40$ (Kepler). Comparando essas figuras com a da arquitetura Maxwell pode ser observado também uma semelha no padrão de comportamento.

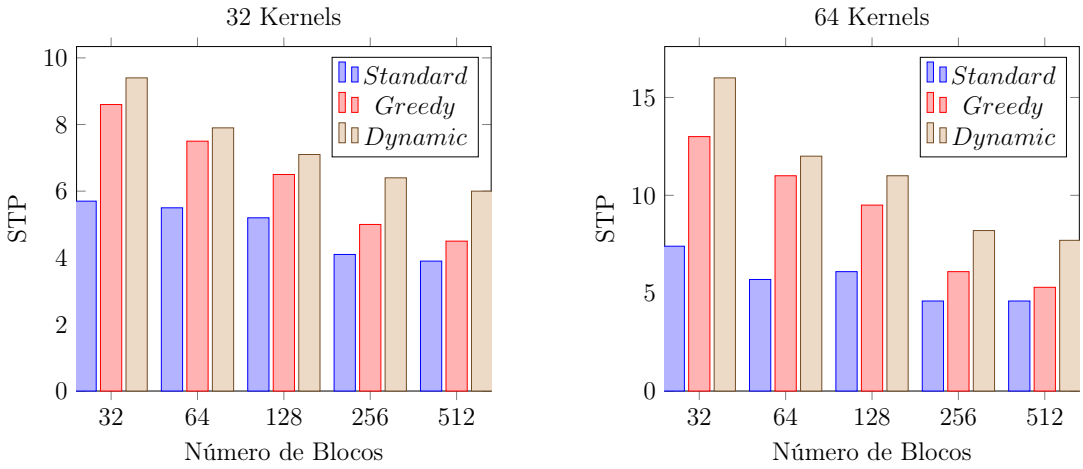

Figura 4.10: Resultado da Portabilidade para $N = 32$ e $N = 64$ na K40 (Kepler).

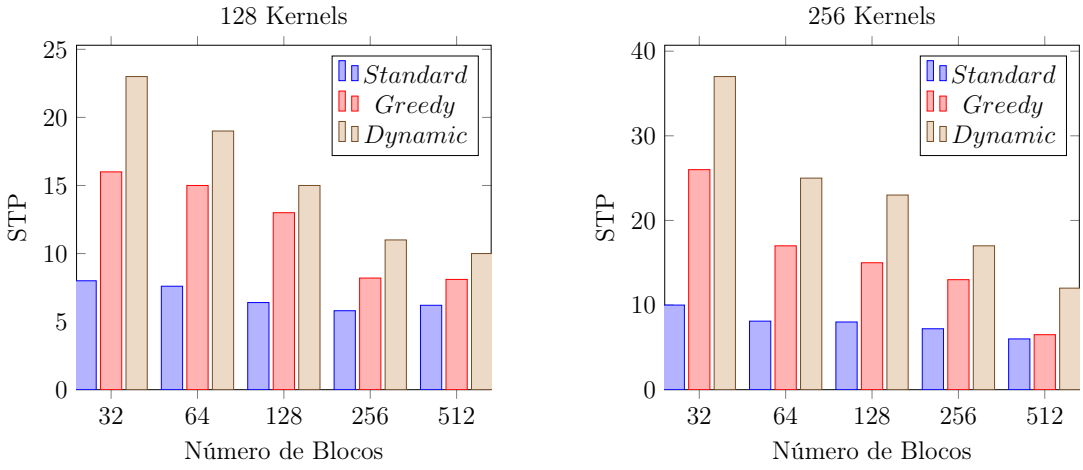

Figura 4.11: Resultado da Portabilidade para $N = 128$ e $N = 256$ na K40 (Kepler).

Capítulo 5

Conclusão

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* 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, o algoritmo do problema da mochila irá buscar novos kernels que podem ocupar melhor os espaços e recursos disponíveis.

Duas implementações do problema da mochila foram implementados, utilizando arquiteturas diferentes de GPUs. As mesmas foram testadas para verificar o quanto o resultado de uma implementação é diferente do outro comparando com as duas arquiteturas. Cada experimento usou de diferentes números de kernels e blocos, onde cada kernel requer um número aleatório de recursos.

Os resultados da proposta mostram que a ordenação adequada leva a um ganho significativo nas métricas $ANTT$ e STP. Além disso, o *overhead* da ordenação é dispersível 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] Awatramani, M.; Zambreno, J.; Rover, D. Increasing gpu throughput using kernel interleaved thread block scheduling. In *2013 IEEE 31st International Conference on Computer Design (ICCD)* (2013), IEEE, pp. 503–506.

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

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

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

[9] 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.

[10] 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.

[11] 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.

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

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

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

[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 sdk. Disponível em `https://developer.nvidia.com/cuda-downloads`.

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

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

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

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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).

[37] 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.

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

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

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

[41] 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.

[42] 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.

[43] 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.

[44] Wende, F.; Cordes, F.; Steinke, T. On improving the performance of multi-threaded 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.

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

[46] 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.

[47] 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.