Maximizando o Uso dos Recursos de GPU Através da Reordenação da Submissão de Kernels Concorrentes

Bernardo Breder[1], Eduardo Charles[1], Rommel Cruz[1], Esteban Clua[1], Cristiana Bentes[2], Lucia Drummond[1]

[1]Instituto de Computação
Universidade Federal Fluminense (UFF), Rio de Janeiro, Brasil

[2]Faculdade de Engenharia
Universidade do Estado do Rio de Janeiro (UERJ), Rio de Janeiro, Brasil

{bbreder,eduardo,rquintanillac,esteban,lucia}@ic.uff.br,
cris@eng.uerj.br

Abstract. *The increasing amount of resources available on current GPUs sparked new interest in the problem of sharing its resources by different kernels. While new generation of GPUs support concurrent kernel execution, their scheduling decisions are taken by the hardware at runtime. The hardware decisions, however, heavily depend on the order at which the kernels are submitted to execution. In this work, we propose a novel optimization approach to reorder the kernels invocation focusing on maximizing the resources utilization, improving the average turnaround time. We model the kernels assignments to the hardware resources as a series of knapsack problems and use dynamic programming approach to solve them. We evaluate our method using kernels with different sizes and resource requirements. Our results show significant gains in the average turnaround time and system throughput compared to the standard kernels submission implemented in modern GPUs.*

Resumo. *O aumento da quantidade de recursos disponíveis nas GPUs modernas despertou um novo interesse no problema do compartilhamento de seus recursos por diferentes kernels. A nova geração de GPUs permite a execução simultânea de kernels, porém ainda são limitadas ao fato de que decisões de escalonamento são tomadas pelo hardware em tempo de execução. Tais decisões dependem da ordem em que os kernels são submetidos para execução, criando execuções onde a GPU não necessariamente está com a melhor taxa de ocupação. Neste trabalho, apresentamos uma proposta de otimização para reordenar a submissão de kernels com foco em: maximizar a utilização dos recursos e melhorar o turnaround time médio. Modelamos a atribuição de kernels para a GPU como uma série de problemas da mochila e usamos uma abordagem de programação dinâmica para resolvê-los. Avaliamos nossa proposta utilizando kernels com diferentes tamanhos e requisitos de recursos. Nossos resultados mostram ganhos significativos no turnaround time médio e no throughput em comparação com a submissão padrão de kernels implementada em GPUs modernas.*

1. Introdução

Unidades de Processamento Gráfico (GPUs) estão cada vez mais populares nos dias de hoje devido ao seu maior desempenho e menor consumo de energia em comparação com as CPUs. GPUs suportam paralelismo maciço a custos relativamente baixos. No caso específico de GPUs da NVIDIA, no modelo de programação CUDA, *kernels* são disparados para a execução e executados por múltiplas *threads*. O programador organiza as *threads* em blocos e a GPU divide os núcleos em conjuntos de Streaming Multiprocessors (SMs). Os blocos de *threads* são criados em tempo de execução e o hardware atribui cada bloco a um SM específico.

Em face do aumento da quantidade de recursos disponíveis em GPUs e os avanços na sua microarquitetura, um novo tipo de problema começa a surgir: como compartilhar de forma eficaz os recursos da GPU por diferentes *kernels*? Muitas aplicações não estão prontas para tirar proveito de todos os recursos. De acordo com Pai *et al.* [Pai et al. 2013], o benchmark Parboil2 usa somente de 20% a 70% dos recursos da GPU Fermi. Adriaens *et al.* [Adriaens et al. 2012] realizaram estudos semelhantes para 12 aplicações reais e mostraram que a maioria delas exibem utilização desequilibrada de recursos GPU.

A fim de melhorar a utilização dos recursos, as nova gerações de GPUs da NVIDIA permitem a execução concorrente de *kernels*. Inicialmente introduzido na arquitetura Fermi, e posteriormente aprimorada na arquitetura Kepler, a execução concorrente de *kernels* permite que blocos de *threads* de diferentes *kernels* possam compartilhar os mesmos recursos da GPU.

A política de escalonamento de blocos de *threads* da GPU é implementada diretamente pelo hardware. Porém, a ordem com que os *kernels* são inseridos nas filas de escalonamento tem grande impacto na ocupação da GPU. Suponha o exemplo da Figura 1, onde quatro *kernels* são submetidos para executar em uma GPU. Cada *kernel* tem um tempo de execução e solicita uma quantidade diferente de recursos. Na Figura 1 (a), mostramos a execução dos kernels considerando a seguinte ordem de submissão: $\{k_1, k_2, k_3, k_4\}$. Podemos observar que de t_0 a t_1 há 40% dos recursos disponíveis e que em t_2 a t_3 há 50%. Entretanto, se a ordem de submissão fosse $\{k_4, k_1, k_3, k_2\}$, obteríamos, conforme ilustrado na Figura 1 (b), um grande aumento no uso dos recursos.

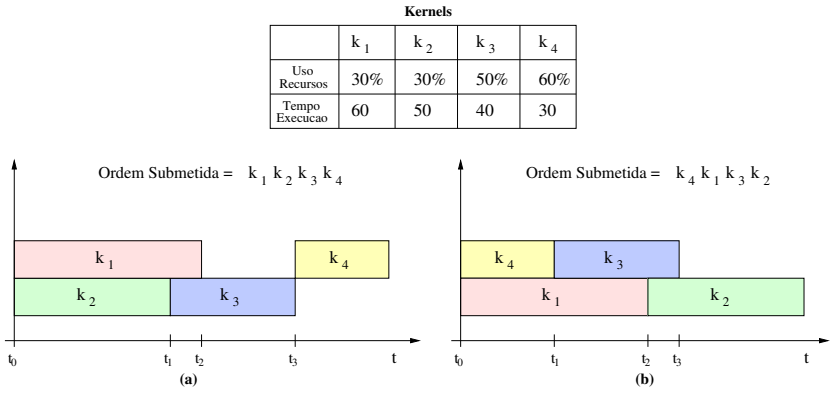

Figura 1. Um conjunto de 4 *kernels* submetidos à GPU em duas ordens diferentes. Em (a), a ordem é $\{k_1, k_2, k_3, k_4\}$. Em (b), a ordem é $\{k_4, k_1, k_3, k_2\}$

Neste trabalho, propomos uma nova estratégia de otimização que determina a me-

lhor ordem para submeter *kernels* para a GPU. A ideia é simular a atribuição de *kernels* para as filas de hardware de modo a maximizar a utilização dos recursos, melhorando o throughput. Nossa proposta modela o problema de selecionar quais *kernels* são melhores para submissão para tirar o máximo proveito dos recursos disponíveis como uma série de problemas da mochila [Martello and Toth 1990]. Modelamos o conjunto dos recursos disponíveis como a capacidade da mochila e o conjunto de núcleos como os itens a serem inseridos na mochila. Os *kernels* têm pesos e valores de acordo com a quantidade de uso de recursos - memória compartilhada, número de registradores e número de *threads* - e o tempo de execução estimado. A solução para o problema da mochila estabelece o subconjunto de *kernels* cujo peso total é menor que os recursos disponíveis e o valor total é tão grande quanto possível. Usamos um algoritmo de programação dinâmica para resolver o problema da mochila e gerar uma nova ordem de submissão dos *kernels*.

Nosso algoritmo de reordenação é focado para uma tendência futura em que GPUs serão dispositivos multiprogramados, com um grande número de *kernels* compartilhando seus recursos. Apresentamos resultados de uma série de experimentos usando diferentes conjuntos de *kernels* com requisitos de recursos aleatórios. A nossa abordagem de reordenação foi capaz de reduzir o tempo médio de execução de 45% para 75% em comparação com a submissão de *kernels* padrão implementada em GPUs modernas. Também mostramos que o overhead referente ao cálculo de uma série de problemas da mochila é insignificante em comparação com os ganhos obtidos.

O restante do artigo é organizado da seguinte forma. A Seção 2 apresenta os trabalhos anteriores. A Seção 3 apresenta uma breve introdução do problema que estamos tratando. A Seção 4 descreve o algoritmo de reordenação proposto. A Seção 5 apresenta os resultados experimentais. Finalmente, na Seção 6, apresentamos nossas conclusões e propostas de trabalhos futuros.

2. Trabalhos Relacionados

Antes de as GPUs possuírem suporte de hardware para a execução simultânea de *kernels*, alguns trabalhos propuseram superar esta limitação através da estratégia de software *merging* de diferentes *kernels* em um só [Peters et al. 2010, Wang et al. 2011, Gregg et al. 2012]. Em uma direção diferente, alguns autores propuseram modificar a granularidade dos *kernels* de modo a aumentar a utilização da GPU [Zhong and He 2014, Ravi et al. 2011, Pai et al. 2013]. Esta estratégia foi chamada de *slicing* por prover o fatiamento dos *kernels* de forma a se obter mais oportunidades de compartilhamento. Houve também alguns esforços em propor a divisão dos recursos da GPU entre os *kernels* concorrentes [Liang et al. 2015, Adriaens et al. 2012]. Estes esforços, no entanto, focaram em apontar melhorias no hardware da GPU para aumentar a sua utilização.

Mais próximo ao nosso trabalho estão as propostas que se concentram em obter melhor utilização com base na ordem em que os *kernels* são invocados no lado do *host*, chamado de *reordenamento de kernels*. Wende *et al.* [Wende et al. 2012] propuseram a reordenação para a arquitetura Fermi, onde a execução simultânea de *kernels* usa apenas uma fila de escalonamento. Eles criaram uma *thread* de escalonamento que organiza a fila de modo *round-robin*, sem considerar a utilização de recursos das *threads*. Os autores foram bem sucedidos no aumento da concorrência intercalando diferentes *streams* de execução na mesma fila, mas o hardware das GPUs modernas possui o

mecanismo Hyper-Q, que implementa essas filas em hardware, possibilitando a execução realmente concorrente de kernels. O nosso trabalho é mais próximo ao trabalho de Li *et al.* [Li et al. 2015] que propõe um esquema de reordenação para arquiteturas de GPU com tecnologia Hyper-Q. Sua abordagem calcula um *score simbiótico*, que tenta co-executar *kernels* com a utilização de recursos complementares e prevê o consumo de energia utilizando o conceito de *rodadas de execução* - rodada de execução simultânea de blocos de *threads* nos SMs. Eles tentam cumprir uma rodada com *kernels* simbióticos usando um algoritmo guloso para o problema *bin-packing* multidimensional. O uso de rodadas de execução simplifica as decisões de escalonamento. Em contraste, a nossa solução de reordenação evita a sincronização implícita na conclusão da rodada, resolvendo um problema mais complexo cada vez que há recursos disponíveis.

3. Definição do Problema

Para uma determinada GPU, D, seja $Ord = \{k_0, k_1, ..., k_{N-1}\}$ a lista de N *kernels* independentes na ordem em que são submetidos para execução em D (k_i é submetido antes de k_{i+1}). Dado que nossa estratégia de reordenação é executada de forma dinâmica, nossa solução pode ser usada em GPUs que estão constantemente recebendo novas demandas de execução. Neste sentido, Ord pode ser atualizada dinamicamente. Considere que NSM é o número de SMs de D, e considere que D possui os recursos $R_j \mid j \in \{0, 1, 2\}$:

- R_0 = tamanho da memória compartilhada por SM;
- R_1 = número de registradores por SM;
- R_2 = número máximo de *threads* por SM.

A capacidade total de cada recurso de D é definida por $W_j = R_j \times NSM \mid j \in \{0, 1, 2\}$. Em um determinado instante de tempo durante a execução, há m *kernels* executando concorrentemente em D, usando uma porcentagem de W_j. Os recursos disponíveis neste instante de tempo são chamados de $W_j^{av} \mid j \in \{0, 1, 2\}$. Toda vez que um *kernel* termina e libera seus recursos, novos *kernels* são selecionados para execução de acordo com a quantidade de recursos disponíveis.

A fim de encontrar uma nova ordem de submissão, Ord^{new}, maximizando a utilização de recursos, modelamos como um problema da mochila 0-1 a seleção de novos *kernels* para submissão. Neste modelo, no início da execução e a cada vez que um *kernel* terminar, D representa a mochila com capacidade igual aos recursos disponíveis e os *kernels* representam os itens a serem empacotados na mochila. Para cada *kernel* k_i, as seguintes informações são coletadas:

- t_i^{est}: tempo de execução estimado;
- sh_i: quantidade de memória compartilhada requisitada por bloco;
- nr_i: número de registradores requisitado por bloco;
- nt_i: número máximo de *threads* disparadas por bloco;
- nb_i: número de blocos.

Com relação ao tempo de execução estimado, há uma série de abordagens para realizar esta previsão com base em diferentes técnicas, tais como baseadas em simulação, *machine learning* e modelos analíticos [Lopez-Novoa et al. 2015]. Outra solução consiste na utilização de medidas de tempo de uma execução anterior da aplicação [Choi et al. 2013]. O estudo e a implementação de tais modelos de desempenho estão fora do escopo deste trabalho.

As quantidades de memória compartilhada, registradores e *threads* podem ser obtidas de forma dinâmica através de diversas ferramentas, tais como o NVIDIA CUDA Profiler [NVIDIA 2016]. Atribuímos para cada kernel k_i um peso $w_{ij}(i \in \{0, ..., N-1\}, j \in \{0, 1, 2\})$, que representa a quantidade de recursos requisitados: $w_{i0} = sh_i \times nb_i$, $w_{i1} = nr_i \times nb_i$, e $w_{i2} = nt_i \times nb_i$. Observe que, se $\exists\ j \mid w_{ij} > R_j$, k_i ainda pode executar uma vez que a GPU pode escalonar um número de blocos de k_i que cabem na GPU e, depois que terminar, escalonar o restante dos blocos. Neste caso, para considerar k_i um item candidato, deve-se ajustar $w_{ij} = R_j$.

Atribuímos também um valor v_i para cada *kernel* k_i que representa a média do percentual dos recursos necessários por unidade de tempo. Kernels com tempos de execução menores terão maiores valores de v_i.

$$v_i = \frac{\frac{sh_i}{R_0} + \frac{nr_i}{R_1} + \frac{nt_i}{R_2}}{3 \times t_i^{est}} \tag{1}$$

Cada problema da mochila consiste em determinar a quantidade de *kernels* cuja soma do lucro é maximizada sem que a soma dos pesos exceda W. O problema da mochila é formulado como a maximização [Martello and Toth 1990]:

$$maximize \sum_{i=0}^{N-1} v_i x_i \tag{2}$$

$$sujeito\ a \sum_{i=0}^{N-1} w_{ij} x_i \leq W_j \mid j \in \{0, 1, 2\} \tag{3}$$

onde $x_i \in \{0, 1\}$, $x_i = 1$ se o item i deve ser incluído na mochila, e $x_i = 0$ caso contrário. Neste problema, a maximização do lucro representa a maximização do uso de recursos.

4. Proposta de Reordenação

Basicamente, nossa proposta de reordenação simula a execução simultânea dos *kernels*. Definimos um número de filas S, tal que S seja igual ao número de filas de escalonamento do hardware. Até S *kernels* podem ser executados simultaneamente na GPU. O algoritmo considera que os *kernels* são escalonados pelo hardware de maneira round-robin enquanto existirem recursos disponíveis.

O Algoritmo 1 descreve a nossa abordagem. De forma geral, ele simula iterativamente o término do próximo *kernel* (que termina mais cedo), e encontra os próximos *kernels* para submeter através da resolução de um problema da mochila com os recursos disponíveis liberados pelo *kernel* que terminou.

Inicialmente, $KernelList$ recebe todos os *kernels* em $Ord = \{k_0, k_1, ..., k_{N-1}\}$, e $W_j^{av} = W_j, \forall\ j \in \{0, 1, 2\}$. A ordem Ord representa a ordem em que os *kernels* são invocados na aplicação. Enquanto houver *kernels* em $KernelList$, o algoritmo funciona da seguinte forma. A função *GetKernelsToSubmit* resolve um problema da mochila 0-1 e provê em *NextSubmitList* a lista de *kernels* em ordem descrescente de valores. Para cada *kernel* k_i em *NextSubmitList*, o algoritmo insere k_i na fila cujo tempo de término

(término do último elemento) é o menor. Em seguida, os recursos necessários para k_i são removidos de W^{av}, e k_i é removido de $KernelList$. O próximo passo é fazer avançar a execução e procurar o próximo término do *kernel* pela função *SelectEarliestKernel*. Esta função procura o *kernel* com o menor t^{est}. A conclusão do *kernel* é simulada através da inserção de seus recursos de volta para W^{av}. No fim, o algoritmo cria Ord^{new} através de uma busca *round-robin* nas filas.

Algoritmo 1 Algoritmo Principal de Reordenação.

```
 1: function KernelReorder(KernelList, W^av)
 2:     while (KernelList not empty) do
 3:         NextSubmitList ← GetKernelsToSubmit(KernelList, W^av)
 4:         for each k_i in NextSubmitList do
 5:             // Simula execução de k_i
 6:             fila_l ← FindEarlyQueue(W^av, Filas)
 7:             Insere k_i em fila_j
 8:             Retém os recursos de k_i removendo-os de W^av
 9:             Remove k_i de KernelList
10:         Earliest ← SelectEarliestKernel(Filas)
11:         // Simula Término do Primeiro
12:         Libera recursos do primeiro a terminar inserindo em W^av
13:     Ord^new ← RoundRobingCross(Stream)
14:     return O^new
```

4.1. Resolvendo o Problema da Mochila para Obter o Conjunto de Kernels

Nossa solução para o problema da mochila multidimensional envolve a linearização dos pesos atribuídos aos *kernels*, de modo que eles sejam representados por um único inteiro. Calculamos um único valor W que representa as três grandezas W_0, W_1 e W_2. Analogamente, os recursos necessários para cada *kernel* também são reduzidos a um único inteiro. Assim, $W = W_0 \times W_1 \times W_2$, e para cada *kernel* k_i, $w_i = (w_{i0} \times W_1 \times W_2) + (w_{i1} \times W_2) + w_{i2}$.

O Algoritmo 2 apresenta o procedimento principal do método dinâmico utilizado para resolver o problema. Nas linhas 2 e 3, as variáveis $SelectedKernels$ e $tempW$ são inicializadas respectivamente com um conjunto vazio de *kernels* e com a capacidade total da GPU. Na linha 4, a função $KnapsackMatrix$ é chamada com os parâmetros $KernelList$ e W^{av}. Esta função retorna uma matriz $M : (N + 1) \times (W + 1)$, onde as linhas representam os *kernels* e as colunas representam uma quantidade linearizada de recursos. Na linha 5, para cada *kernel* k_i, se houver recursos restantes, avalia-se a possibilidade de incluir k_i através de consulta à matriz M. Na linha 7, o algoritmo testa se k_i foi selecionado pelo método dinâmico. Se o valor da linha i é diferente do valor prévio na coluna $tempW$ (o que significa que a programação dinâmica selecionou k_i), ele é subtraído de w_i e k_i é incluído na lista $SelectedKernels$ (linhas 8 e 9). No final, a lista $NextSubmitList$ recebe os elementos de $SelectedKernels$ ordenados em ordem descrescente de valor (linha 10).

A função $KnapsackMatrix$, que retorna a matriz M, é detalhada no Algoritmo 3. A matriz M é inicializada na linha 3 e pode ser atualizada nas linhas 7 ou 9. Na linha 7, uma célula de M é copiada para outra, o que significa que o *kernel* k_i não é selecionado para ser incluído na mochila, porque excede a capacidade restante da mochila. Caso contrário, se ele gerar algum lucro, ele pode ser incluído na mochila (linha 9).

Algoritmo 2 Programação Dinâmica - Procedimento Principal

```
1: function GetKernelsToSubmit(KernelList, W^av)
2:     SelectedKernels ← ∅
3:     tempW ← W^av
4:     M ← KnapsackMatrix(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
```

Algoritmo 3 Programação Dinâmica - Construção da Matriz M

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

5. Resultados Experimentais

5.1. Ambiente Computacional

Utilizamos duas GPUs diferentes em nossos experimentos: TITAN X (arquitetura Maxwell) e Tesla K40 (arquitetura Kepler). Suas principais características estão apresentadas na Tabela 1. Os *kernels* foram implementados utilizando CUDA 7.5.

	TITAN X	**K40**
Número de cores	3.072	2.880
Clock do Core	1000 MHz	745 MHz
RAM	24GB	12GB
Bandapassante de Memória	336,5 GB/s	288 GB/s
Capability	5.2	3.5
Número de SMs	24	15
Memória Compartilhada por SM	96KB	48KB
Número de Registradores por SM	64K	64K
Máx Num de Threads por SM	2048	2048
Arquitetura	Maxwell	Kepler
Performance no Pico	6,6 TFLOPs	4,2 TFLOPS

Tabela 1. Configurações das GPUs utilizadas.

5.2. Metodologia

Como a execução concorrente de *kernels* é uma característica relativamente nova em GPUs da NVIDIA, os programas em CUDA tendem a explorar o paralelismo de *threads*

mais do que o paralelismo entre *kernels*. Acreditamos que no futuro, as GPUs vão funcionar como dispositivos multiprogramados, mas até o momento as aplicações de suites de *benchmarks*, como Rodinia, Parboil ou CUDA SDK, possuem um número muito pequeno de *kernels* independentes. Encontrar uma boa configuração de *kernels* com requerimentos de recursos realmente constrastantes é muito complicado nestes *benchmarks*. Portanto, decidimos avaliar nossa proposta de reordenação utilizando benchmarks sintéticos, em que podemos variar bastante e de forma controlada o número de *kernels* concorrentes e o requerimento de recursos.

Implementamos um gerador de *kernel* que cria conjuntos de N *kernels* independentes em 32 CUDA *streams* para um dispositivo D que possui os recursos R_0, R_1 e R_2. Os valores de N e $R_j \mid j \in \{0, 1, 2\}$ são passados como parâmetros.

A estrutura geral de cada *kernel* sintético k_i é de um código CUDA que executa um conjunto de operações aritméticas em uma matriz de números inteiros alocados na memória compartilhada. O número de blocos de k_i e suas necessidades de recursos, sh_i, nt_i e t_i^{est}, são criados randomicamente. Nos nossos experimentos, não estressamos o valor de nr_i, mas o mantivemos igual a zero.

Criamos diferentes experimentos com N variando em $\{32, 64, 128, 256\}$. Para cada valor de N, variamos também o número máximo e mínimo de blocos por kernel. O número mínimo de blocos é atribuído como 4 e o número máximo é aleatório mas limitado pelo valor de NB que varia em $\{32, 64, 128, 256, 512\}$.

Para cada experimento com N *kernels*, realizamos 50 execuções diferentes de modo a reduzir o impacto da aleatoriedade nos resultados. O gerador de *kernels* e o algoritmo de reordenação foram implementados em Java 1.7. Comparamos a nossa abordagem de reordenação com um cenário em que os *kernels* são submetidos à execução na mesma ordem em que foram invocadas, que chamamos de abordagem *Padrão*.

As métricas de avaliação utilizadas são: Average Normalized Turnaround Time (ANTT) e System Throughput (STP) propostas por Eyerman and Eeckhout em [Eyerman and Eeckhout 2008]. ANTT é uma métrica quanto menor melhor e mede o *turnaround time* médio normalizado pela execução do *kernel* sem concorrência. STP é uma métrica quanto maior melhor e mede a quantidade de tarefas completadas por unidade de tempo. Dado que o *turnaround time* de um *kernel* k_i é representado por TT_i, e o *turnaround time* de k_i quando executado sozinho é TT_i^{SP}:

$$ANTT = \frac{1}{N} \sum_{i=0}^{N-1} \frac{TT_i}{TT_i^{SP}} \qquad (4) \qquad\qquad STP = \sum_{i=0}^{N-1} \frac{TT_i^{SP}}{TT_i} \qquad (5)$$

5.3. Análise dos Resultados

As Figuras 4 a 7 mostram os resultados de ANTT para diferentes valores de N e diferentes valores de NB. Podemos observar nestas figuras que reordenar a submissão de *kernels* tem um impacto significativo sobre o *turnaround time* médio do conjunto de *kernels*. As reduções em ANTT variam de 45% a 75%. Podemos observar também que o conjunto de *kernels* com $N = 256$ produziu ganhos mais pronunciados. Isso ocorre porque há mais itens para serem inseridos na mochila e o algoritmo tem mais possibilidades de enchê-la.

A Tabela 2 mostra os resultados de STP para as mesmas variações de N e NB.

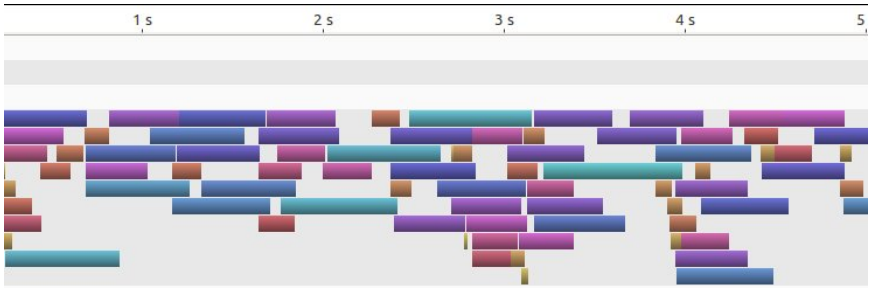

Figura 2. Profile de execução de 128 kernels com submissão padrão.

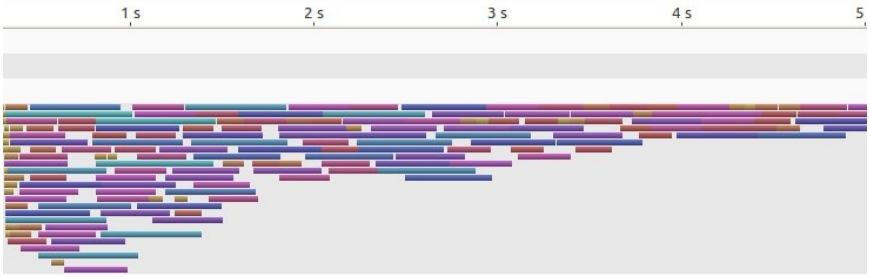

Figura 3. Profile de execução de 128 kernels com submissão reordenada.

Observamos nesta tabela que o *throughput* é consideravelmente maior quando nosso esquema de reordenação é utilizado. Nossa abordagem aumenta o STP de 1,6 a 3,5 vezes, o que significa que ele está melhorando a utilização da GPU. Ilustramos essa utilização nas Figuras 2 e 3 que mostram um corte dos resultados (entre de 0 a 5s) do CUDA Profiler para $N = 128$ e $NB = 32$. Podemos observar que a execução padrão permite um máximo de 10 núcleos executando simultaneamente, enquanto que a execução com reordenação permite até 24 *kernels* simultâneos e um consequente aumento do uso de recursos.

	STP							
	32 Kernels		64 Kernels		128 Kernels		256 Kernels	
NB	Padrão	Reord.	Padrão	Reord.	Padrão	Reord.	Padrão	Reord.
32	11	18	14	25	16	39	21	54
64	7.6	13	9.3	19	10	29	12	43
128	5.6	10	8.0	14	8.8	23	9.6	28
256	5.0	8.2	5.5	10	6.6	17	7.1	22
512	4.2	6.6	5.0	8.9	5.4	12	6.5	16

Tabela 2. Resultados de STP para diferentes valores de N and NB.

5.4. Overhead

A reordenação é executada estaticamente antes que os *kernels* sejam submetidos à GPU. No entanto, uma vez que o cálculo da reordenação envolve a solução de um problema de otimização combinatória, avaliamos a overhead do algoritmo de reordenação. A Tabela 3 mostra a overhead de nossa abordagem de reordenação para $N \in \{32, 64, 128, 256\}$ e $NB \in \{32, 64, 128, 256, 512\}$. O overhead é calculado comparando o tempo de execução do algoritmo de reordenação com o ganho absoluto no *turnaround time* médio gerado por reordenar os *kernels*. O overhead mostra a porcentagem de nossos ganhos que foi gasta em computar a solução.

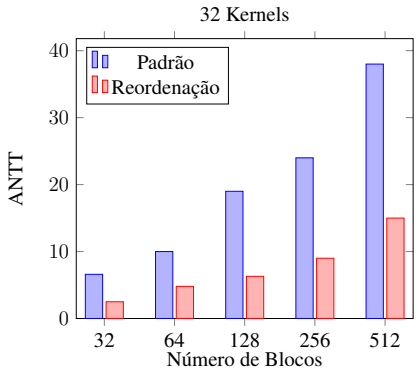

Figura 4. Resultados de ANTT para N=32 na TITAN X.

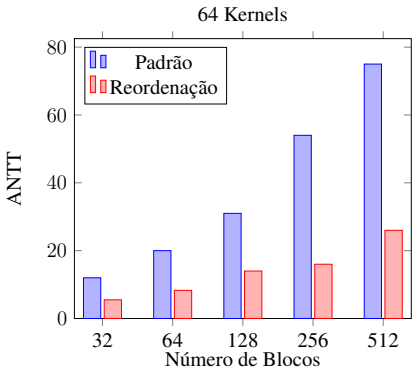

Figura 5. Resultados de ANTT para N=64 na TITAN X.

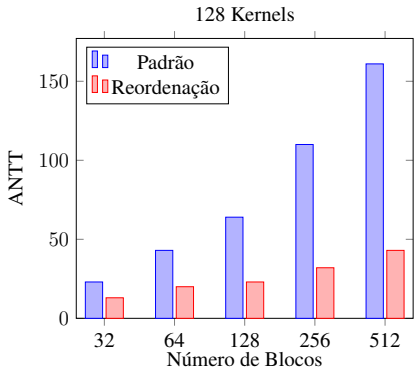

Figura 6. Resultados de ANTT para N=128 na TITAN X.

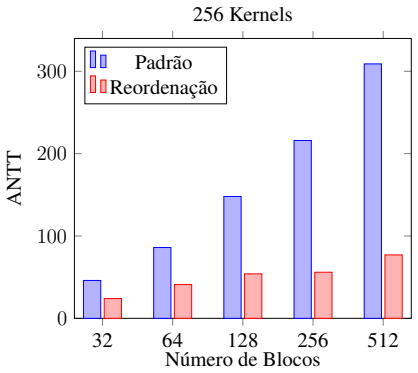

Figura 7. Resultados de ANTT para N=256 na TITAN X.

Podemos observar na Tabela 3 que os overheads são baixos. Gastamos de 0,1% para 6,3% dos ganhos para calcular a reordenação. Também podemos observar que os maiores overheads foram encontrados com um pequeno número de *kernels* e blocos. Isso ocorre porque a pequena quantidade de computação leva a ganhos menores no tempo médio. Para maior número de *kernels* e blocos, o overhead de resolver os problemas da mochila torna-se insignificante.

	Overhead			
NB	32 Kernels	64 Kernels	128 Kernels	256 Kernels
32	6.3	2.1	2.1	2.5
64	2.7	1.0	1.0	1.3
128	0.5	0.5	0.5	0.8
256	0.4	0.2	0.2	0.4
512	0.2	0.2	0.2	0.2

Tabela 3. Overhead da Reordenação.

5.5. Portabilidade

Nosso esquema de reordenação é uma solução de software que pode ser aplicada a diferentes arquiteturas da NVIDIA que permitam a existência de *kernels* concorrentes. Avaliamos também nossa abordagem usando uma GPU baseada na arquitetura Kepler. Apresentamos nas Tabelas 4 a 7 os resultados de ANTT e STP para a GPU K40. Nossa

solução atinge reduções substanciais em ANTT e melhorias em STP quando comparada com a submissão padrão. Obtivemos de 31% a 73% de redução em ANTT, e STP foi melhorado de 1,1 a 3,2 vezes. Comparando com os resultados da TITAN-X, observamos ganhos semelhantes.

	ANTT		STP	
NB	Padrão	Reord	Padrão	Reord
32	17	8.5	6.2	8.7
64	18	10	5.0	7.9
128	28	17	4.5	6.9
256	50	34	4.1	5.8
512	53	35	4.1	4.5

Tabela 4. Execução com 32 kernels.

	ANTT		STP	
NB	Padrão	Reord	Padrão	Reord
32	28	12	7.1	15
64	49	16	6.3	12
128	68	22	4.7	9.4
256	98	25	4.9	7.6
512	116	35	5.0	7.5

Tabela 5. Execução com 64 kernels.

	ANTT		STP	
NB	Padrão	Reord	Padrão	Reord
32	56	23	8.4	21
64	91	32	6.9	18
128	133	40	5.6	13
256	166	59	5.9	11
512	220	76	5.6	10

Tabela 6. Execução com 128 kernels.

	ANTT		STP	
NB	Padrão	Reord	Padrão	Reord
32	115	50	9.2	29
64	183	64	7.4	22
128	261	85	6.9	17
256	355	108	6.1	13
512	419	143	6.3	12

Tabela 7. Execução com 128 kernels.

6. Conclusões

Como o hardware da GPU continua evoluindo, oferecendo uma quantidade cada vez maior de recursos, é muito provável que, no futuro, a GPU avance para um dispositivo multiprogramado. Neste cenário, o compartilhamento dos recursos da GPU é a chave para melhorar o throughput. Neste trabalho, propomos uma estratégia de reordenação dos *kernels* submetidos para executar na GPU. Nossa ideia é resolver uma série de problemas da mochila para encontrar os *kernels* que maximizam a utilização GPU. Toda vez que há recursos disponíveis, o algoritmo modela a quantidade de recursos como a capacidade da mochila e tenta encher a mochila com os *kernels* que levam ao máximo proveito desses recursos disponíveis, favorecendo os *kernels* com menor tempo de execução.

Apresentamos uma série de experimentos utilizando *kernels* sintéticos que requerem diferentes quantidades de recursos. Nossos resultados mostram que a reordenação proporciona ganhos significativos no tempo médio de *turnaround* e no *throughput* do sistema em comparação com a submissão de *kernels* padrão implementada em GPUs modernas. O overhead do algoritmo de reordenação é insignificante em comparação com os ganhos no tempo médio de execução. Também mostramos que a nossa abordagem proporciona ganhos semelhantes nas GPUs Kepler e Maxwell. Futuramente, pretendemos incorporar o nosso esquema de reordenação no serviço de multi-processamento NVIDIA (MPS). O uso de *benchmarks* da litaretura é um passo futuro importante para validar o modelo em diferentes cenários.

Agradecimentos

Este trabalho foi parcialmente financiado com recursos da NVIDIA, CNPq, CAPES e FAPERJ.

Referências

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

Choi, H. J., Son, D. O., Kang, S. G., Kim, J. M., Lee, H.-H., and Kim, C. H. (2013). An efficient scheduling scheme using estimated execution time for heterogeneous computing systems. *The Journal of Supercomputing*, 65(2):886–902.

Eyerman, S. and Eeckhout, L. (2008). System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53.

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

Li, T., Narayana, V. K., and El-Ghazawi, T. (2015). A power-aware symbiotic scheduling algorithm for concurrent GPU kernels. In *The 21st IEEE International Conference on Parallel and Distributed Systems (ICPADS)*.

Liang, Y., Huynh, P., Rupnow, K., Goh, R., and Chen, D. (2015). Efficient GPU spatial-temporal multitasking. *IEEE Trans. on Parallel and Distributed Systems*, 26:748–760.

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

Martello, S. and Toth, P. (1990). *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc.

NVIDIA (2016). Cuda Profiler. http://docs.nvidia.com/cuda/profiler-users-guide.

Pai, S., Thazhuthaveetil, M. J., and Govindarajan, R. (2013). Improving GPGPU concurrency with elastic kernels. In *ACM SIGPLAN Notices*, volume 48, pages 407–418.

Peters, H., Koper, M., and Luttenberger, N. (2010). Efficiently using a CUDA-enabled GPU as shared resource. In *IEEE 10th International Conference on Computer and Information Technology (CIT)*, pages 1122–1127. IEEE.

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

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

Wende, F., Cordes, F., and Steinke, T. (2012). On improving the performance of multi-threaded CUDA applications with concurrent kernel execution by kernel reordering. In *Symp. on Application Accelerators in High Performance Computing (SAAHPC),74-83*.

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