Resolvendo o Problema de Reordenação de Kernels Concorrentes na GPU Utilizando o Problema da Mochila

Rommel Cruz, Bernardo Breder, Pablo Carvalho, Eduardo Charles, Esteban Clua, Lucia M. A.Drummond
Instituto de Computação
Universidade Federal Fluminense (UFF)
{rquintanillac,bbreder,pablo,eduardo,esteban,lucia}@ic.uff.br

Cristiana Bentes
Departamento de Engenharia de Sistemas e Computação
Universidade do Estado do Rio de Janeiro
cris@eng.uerj.br

RESUMO

O compartilhamento de GPUs é uma tendência importante em computação de alto desempenho, principalmente em ambientes de GPUs virtuais, que procuram atender demandas de clientes num ambiente de nuvem. Neste cenário, as GPUs precisam ser capazes de lidar eficientemente com uma carga variada de aplicações e fornecer throughput compatível. Neste trabalho, tratamos do escalonamento dos *kernels* para a execução na GPU. Apresentamos estratégias para definir a ordem em que os *kernels* são escalonados para execução. Como este é um problema de otimização, ele pode ser modelado como um problema da mochila 0-1. Neste trabalho, propomos o uso de uma heurística gulosa para resolver o problema da mochila. Avaliamos os ganhos e custos do método guloso com relação ao método de programação dinâmica. Nossos resultados mostram que o método guloso, obteve bons resultados de desempenho apresentando um overhead de execução menor, podendo ser uma solução viável em ambientes de grande escala.

PALAVRAS CHAVE. GPU, Ordenação de Kernels, Problema da Mochila.

Tópicos (Algoritmos Paralelos em CPU & GPU em Pesquisa Operacional)

ABSTRACT

Sharing GPUs is an important trend in high-performance computing, especially true in virtual GPU environments that seek to meet the demands of customers in a cloud environment. In this scenario, GPUs need to be able to efficiently handle a variety of application loads and provide compatible throughput. In this work, we discuss the scheduling of kernels for execution on the GPU. More specifically, we present strategies to define the order in which the kernels are scheduled for execution. As this is an optimization problem, it can be modeled as a 0-1 knapsack problem. In this work, we propose the use of a greedy heuristic to solve the knapsack problem. We evaluated the gains and costs of the greedy method compared to the dynamic programming method. Our results show that the greedy method obtained good performance results with a lower execution overhead, which can be a viable solution in large scale environments.

KEYWORDS. GPU, Kernel Reordering, Knapsack Problem.

Topics (Parallel CPU & GPU Algorithms for Operational Research)

1. Introdução

Nos últimos anos, clusters heterogêneos, equipados com CPUs de uso geral e unidades de processamento gráfico (GPUs), têm sido amplamente adotados em ambientes computacionais de alto desempenho em larga escala. Isto tem sido motivado principalmente pela sua relação custo-desempenho-potência favorável, bem como pela disponibilidade de modelos de programação relativamente simples como CUDA da NVIDIA. Esta tecnologia tem demonstrado desempenho considerável para a solução de muitos problemas científicos com altos requisitos computacionais. Em consequência, alguns dos supercomputadores mais poderosos do mundo são atualmente construídos utilizando clusters heterogêneos CPU-GPU.

Tradicionalmente, clusters heterogêneos CPU-GPU têm sido construídos com uma ou mais GPUs acopladas a um nó do cluster. Em muitas situações, no entanto, esse tipo de configuração gera uma baixa utilização dos recursos computacionais disponíveis. Aplicações possuem, em geral, dificuldade de explorar adequadamente todo o poder computacional das GPUs modernas, sendo bastante improvável que todas as GPUs do cluster sejam usadas 100% do tempo.

Uma possível solução para esta questão está em diminuir a quantidade de GPUs do cluster e permitir que as aplicações possam **compartilhar** as GPUs existentes. Dentro do conceito de computação verde, esta é uma solução mais eficiente em termos de consumo de energia. O compartilhamento das GPUs pode ser realizado através de ferramentas de virtualização [Duato et al. 2010] que criam um *pool* de servidores de GPU acessível por todos os nós do cluster. Neste cenário, entretanto, as GPUs precisam ser capazes de lidar eficientemente com uma carga variada de aplicações e fornecer throughput compatível para serem usadas como dispositivos compartilhados.

GPUs modernas, no entanto, não são dispositivos multiprogramados como são as CPUs atuais [Liang et al. 2015]. GPUs da NVIDIA até possuem suporte de hardware para execução simultânea de *kernels*, graças a tecnologia de Hyper-Q, onde *kernels* podem ser disparados em *streams* diferentes que podem executar concorrentemente na GPU e a ferramenta CUDA MPS[1] permite a execução concorrente de *kernels* de aplicações diferentes. Ainda assim, questões importantes sobre o escalonamento dos *kernels* para a execução não são consideradas nestes mecanismos. A ordem em que os *kernels* são escalonados para execução tem grande impacto no throughput do sistema, na sua taxa de ocupação e na utilização dos recursos da GPU.

A ordem de escalonamento de aplicações é um problema bastante explorado em ambientes baseados em CPUs, porém pouco explorado no ambiente de GPU [Li et al. 2015] [Adriaens et al. 2012]. Dado um conjunto de *kernels* $\{k_0, k_1, ..., k_{N-1}\}$, qual a melhor ordem para submetê-los de forma a maximizar o throughput da execução? Este é um problema de otimização que foi modelado em [Breder et al. 2016] como um problema da mochila 0-1. Neste modelo, no início da execução e a cada vez que um *kernel* terminar, a GPU representa a mochila com capacidade igual aos recursos disponíveis e os *kernels* representam os itens a serem empacotados na mochila. A solução proposta em [Breder et al. 2016], entretanto, propõe o uso de um método de programação dinâmica para resolver o problema da mochila. Este método, porém, pode gerar um alto consumo de memória, caso o problema seja muito grande.

Neste trabalho, propomos o uso de uma heurística gulosa para resolver o problema da mochila que modela a questão da ordem de submissão de *kernels*. Avaliamos os ganhos e os custos da heurística gulosa com relação ao método mais complexo de programação dinâmica. Embora nossos resultados mostrem que o método guloso obteve resultados de throughput ligeiramente inferiores aos do método de programação dinâmica, ele se mostrou uma alternativa bastante viável para a reordenação de kernels, quando o número de *kernels* é muito grande, devido à sua simplicidade.

2. Definição do Problema

Seja uma GPU um dispositivo composto de um conjunto de Streaming Multiprocessors (SMs), cada um sendo composto por um conjunto de elementos de processamento chamados de Scalar

[1] https://docs.nvidia.com/deploy/pdf/CUDAMultiProcessService-Overview.pdf

Processors (SPs), uma memória compartilhada e um conjunto de registradores tal qual mostra a Figura 1. Os SPs podem ser vistos como núcleos de processamento (*cores*). A quantidade de SMs é dada por nSM e a capacidade de memória compartilhada de cada SM_i é dada por S_i e o tamanho do conjunto de registradores é dado por R_i.

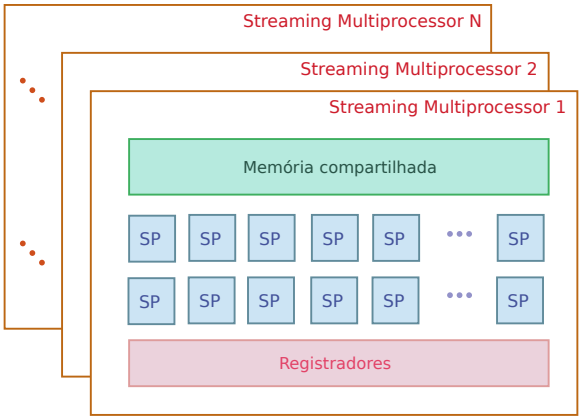

Figura 1: Diagrama de uma GPU padrão

A GPU executa uma função especial disparada pela CPU que é chamada de *kernel*. Um *kernel* executa em paralelo na GPU utilizando diferentes threads. A GPU suporta um grande número de threads com paralelismo finamente granulado. As threads são organizadas em blocos de threads. Os blocos de threads são executados na GPU através da atribuição de um determinado número de blocos para serem executados em um SM. O número máximo de threads suportado por um SM_i é dado por T_i.

Seja W a capacidade da GPU em termos de memória compartilhada, registradores e número de threads. Em um determinado instante, há m *kernels* executando concorrentemente na GPU e utilizando uma porcetagem de W. Neste ponto, a quantidade de recursos disponíveis na GPU é dada por W^{av}. Sempre que um *kernel* termina, outros *kernels* são escolhidos para executar de acordo com a quantidade de recursos disponíveis para execução.

Para cada *kernel* não submetido k_i, os seguintes dados são coletados:

- t_i^{est}: tempo estimado de execução
- b_i: o número de blocos
- sh_i: a quantidade de memória compartilhada por bloco
- nr_i: o número de registradores por bloco
- nt_i: o número de threads por bloco

A quantidade de memória compartilhada, número de registradores, número de threads e o número de blocos podem ser obtidos através de ferramenta de *Profiler* da NVIDIA[2]. O tempo estimado da execução de um *kernel* pode ser obtido de uma execução anterior ou por técnicas de estimativa de tempo [Lopez-Novoa et al. 2015].

Considerando que um conjunto de N kernels, $\{k_1, ..., k_N\}$, foi submetido para execução, o problema da ordem de submissão consiste em determinar uma ordenação dos *kernels* de modo a maximizar o throughput de execução. O throughput, chamado de STP (System Throughput), é uma métrica quanto-maior-melhor que mede o número de tarefas concluídas por unidade de tempo.

[2]http://docs.nvidia.com/cuda/profiler-users-guide

O valor do throughput depende do tempo total de execução dos kernels, chamado *turnaround time*. Seja o *turnaround time* do *kernel* k_i quando executado sozinho TT_i^S e o *turnaround time* do *kernel* k_i executando concorrentemente com outros *kernels* TT_i. O valor do throughput, segundo formulado por [Eyerman and Eeckhout 2008] é dado pela equação (1)

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

Além disso, a ordenação dos *kernels* deve também minimizar o *turnaround time* médio da execução dos kernels. O *turnaround time* médio normalizado, chamado ANTT (Average Normalized Turnaroud Time) é dado pela equação (2). Essa métrica indica em média o quanto o *kernel* k_i foi prejudicado pela concorrência com relação à execução em um ambiente não concorrente.

$$ANTT = \frac{1}{N} \sum_{i=1}^{N} \frac{TT_i}{TT_i^{Alone}} \tag{2}$$

3. Modelagem para o Problema da Mochila 0-1

Dados n itens a serem inseridos em uma mochila com capacidade c, onde cada item i possui um peso w_i e um valor de utilidade v_i, o problema da mochila 0-1 consiste em escolher quais itens serão alocados na mochila de modo que obtenha-se o maior lucro com os valores sem exceder a capacidade c. Considerando a variável de decisão x_i de modo que, se o item i é alocado na mochila, então $x_i = 1$, caso contrário, então $x_i = 0$, o problema pode ser definido pela equação 3.

$$\begin{aligned} maximizar\ x &= \sum_{i=1}^{N} v_i x_i \\ sujeito\ a \sum_{i=1}^{N} w_i x_i &\leq c \end{aligned} \tag{3}$$

Adaptando este problema clássico para o problema da ordem de submissão de *kernels*, definimos que os itens a serem inseridos na mochila são os *kernels* a serem submetidos e a capacidade da mochila c é dada pela capacidade da GPU W, onde , $W = W_0 \times W_1 \times W_2$ e $W_0 = S_i \times nSM$, $W_1 = R_i \times nSM$ e $W_2 = T_i \times nSM$. Há três grandezas diferentes para definir a quantidade de recursos requerida $w_{i0} = sh_i \times nb_i$, $w_{i1} = nr_i \times nb_i$, and $w_{i2} = nt_i \times nb_i$. Neste caso, para definir o peso de cada *kernel* normalizamos para um único valor.

O valor v_i de cada *kernel* é definido pela média das porcentagens dos recursos do item pelo tempo estimado de execução, conforme mostra a equação 4. A divisão pelo tempo estimado de execução é realizada para priorizar *kernels* menores, já que em escalonamento de processos, priorizar os menores leva a um menor tempo médio de espera na fila.

$$v_i = \frac{\frac{w_{i0}}{W_0} + \frac{w_{i1}}{W_1} + \frac{w_{i2}}{W_2}}{3 \times t_i^{est}} \tag{4}$$

4. Algoritmo para Ordenação de Kernels

O Algoritmo 1 realiza a ordenação dos *kernels*. Os parâmetros de entrada são os *kernels* para serem ordenados e a quantidade de recursos disponíveis na GPU. O Algoritmo consiste em um *loop* (linha 2) enquanto todos os *kernels* não foram submetidos, onde cada instante T_β corresponde a uma iteração desse *loop*. No instante T_β, é solicitado para o procedimento $GetkernelsToSubmit$ (linha 3) que os *kernels* selecionados saiam da fila de espera e entrem em execução. Para cada *kernel* k_i selecionado (linha 4), será enfileirado os NQ *kernels* k_i na fila mais curta (linha 5 e 6),

decrementado w_i dos recursos disponíveis W^{av} (linha 7) e removido o k_i da lista de *kernels* a serem submetidos (linha 8). Para finalizar, 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, as filas estarão preenchidas com os *kernels* selecionados e serão retornados os *kernels* na ordem das filas seguindo o modo *Round-Robin*.

Algoritmo 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)
```

O procedimento $GetKernelsToSubmit$ recebe como entrada os *kernels* que ainda não foram submetidos e a quantidade de recursos disponíveis no instante T_β. Esse procedimento irá resolver o problema da mochila com as adaptações descritas utilizando o Algoritmo Guloso e o Algoritmo de Programação Dinâmica (conforme prosposto em [Breder et al. 2016]).

4.1. Algoritmo Guloso

A heurística gulosa é uma técnica simples que pode ser aplicada a uma grande variedade de problemas. A grande maioria destes problemas possui um conjunto de entradas e qualquer subconjunto que satisfaça as restrições do problema, representa uma solução viável. Deseja-se então encontrar uma solução viável que maximize ou minimize uma dada função objetivo.

A solução do Problema da Mochila utilizando a heurística gulosa, chamada aqui de $Greedy$, é apresentada no Algoritmo 2. Neste algoritmo, 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 esta ordem, para cada *kernel* k_i (linha 5), se houver recursos suficientes $tempW$ para k_i (linha 6), os recursos w_i do *kernel* serão decrementados (linha 7) e ele será selecionado (linha 8).

Algoritmo 2 Heurística Gulosa para a função $GetKernelsToSubmit$

```
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
```

4.2. Programação Dinâmica

Para resolver o problema da mochila utilizando programação dinâmica, é preciso decompor o problema em problemas menores. Chamamos aqui o método de programação dinâmica de $Dynamic$. Para isso, propomos usar uma fórmula recursiva para percorrer todas as possíveis decisões. A equação 5 [Martello and Toth 1990] mostra a equação que denota o valor máximo obtido na mochila ao combinar os valores dos pesos w_i com a inclusão dos *kernels* através de x_i.

$$OPT(i,w) = \max\left(\sum_{i=1}^{N} v_i x_i\right)$$
$$sujeito\ a \sum_{i=1}^{N} w_{ij} x_i \leq W_j \mid j \in \{0,1,2\} \tag{5}$$

A partir da equação 5, o cálculo de recorrência para o valor máximo obtido na mochila pode ser expressa pela equação 6.

$$OPT(i,w) = \begin{cases} 0, \text{se } i = 0 \\ OPT(i-1,w), \text{se } w < w_i \\ \max(OPT(i-1,w), v_i + OPT(i-1, w-w_i)) \end{cases} \tag{6}$$

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

Com a equação de recorrência 6, 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.

Algoritmo 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
```

4.3. Vantagens e Desvantagens

A resolução do problema da mochila usando Programação Dinâmica provê solução ótima, enquanto que usando a heurística gulosa produz uma boa solução, mas não necessariamente ótima. A Programação Dinâmica deve ser utilizada quando a diferença entre uma solução boa e ótima é impactante para a aplicação. Esse método garante a solução ótima para o problema, porque leva em consideração o contexto global, mas requer a construção e o preenchimento de uma matriz M cuja dimensão é o número de itens a ser colocados na mochila. Dependendo da quantidade de itens e da capacidade da mochila, o método pode se tornar inviável devido ao alto consumo de memória.

Em função da simplicidade da heurística gulosa, a complexidade de processamento e de memória não costuma ser um problema. A estratégia utilizada na ordenação dos itens se trata de uma heurística que usa a razão entre o valor e o peso para representar uma boa solução num contexto local.

5. Resultados Experimentais

5.1. Ambiente Computacional

Utilizamos em nossos experimentos a GPU TITAN X (arquitetura Maxwell). Suas principais características são apresentadas na Tabela 1. Os *kernels* foram implementados utilizando CUDA 7.5.

	TITAN X
Número de cores	3.072
Clock do Core	1000 MHz
RAM	24GB
Bandapassante de Memória	336,5 GB/s
Capability	5.2
Número de SMs	24
Memória Compartilhada por SM	96KB
Número de Registradores por SM	64K
Máx Num de Threads por SM	2048
Arquitetura	Maxwell
Performance no Pico	6,6 TFLOPs

Tabela 1: Configurações das GPUs utilizadas.

5.2. Metodologia

A execução de *kernels* concorrentes é uma funcionalidade razoavelmente nova em GPUs da NVIDIA, disponível de forma rudimentar desde a arquitetura Fermi. Por este motivo, uma gama muito grande de programas para a GPU foram desenvolvidos explorando somente o paralelismo de granularidade fina entre *threads* de um mesmo *kernel* ao invés de explorar o paralelismo entre kernels. Os *benchmarks* existentes para CUDA na literatura, por exemplo, Rodinia [Che et al. 2009], Parboil [Stratton et al. 2012] e CUDA SDK[3], quase não exploram o paralelismo entre kernels. Decidimos, portanto, avaliar a reordenação utilizando benchmarks sintéticos, possibilitando um controle mais refinado do número de *kernels* e de suas especificações.

Para cada experimento, criamos conjunto de N *kernels* independentes 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 $\{32, 64, 128, 256, 512\}$. Além disso, as necessidades de recursos de um kernel, sh_i, nt_i e t_i^{est}, também são criadas randomicamente. A estrutura geral de cada *kernel* k_i sintético consiste num conjunto de códigos CUDA que realizam algumas operações aritméticas em um vetor de inteiros alocado na memória compartilhada. Uma quantidade de registradores também é alocada, 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 reordenação. 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.

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 as duas estratégias de reordenação que resolvem o problema da mochila com as abordagens gulosa ($Greedy$) e com programação dinâmica ($Dynamic$). Os resultados das reordenação desses dois métodos são também comparados com o resultado dado pela ordenação padrão do programa ($Standard$), onde os N *kernels* são ordenados na ordem crescente pelo índice. Por exemplo, para $N = 4$, a reordenação $Standard$ corresponderá a $Ord = \{k_1, k_2, k_3, k_4\}$.

5.3. Análise dos Resultados

As Figuras 2 a 4 mostram o *profile* da execução dos kernels. As figuras mostram as filas de tarefas no eixo y e a linha de tempo no eixo x com uma amostra de 2 segundos, onde cada caixa retangular do *trace* representa a execução de um kernel. Pode-se observar que muitos kernels, representados por caixas retangulares, são executados concorrentemente no inicio da Figura 4 e poucos são executados na Figura 2.

[3]https://developer.nvidia.com/cuda-downloads

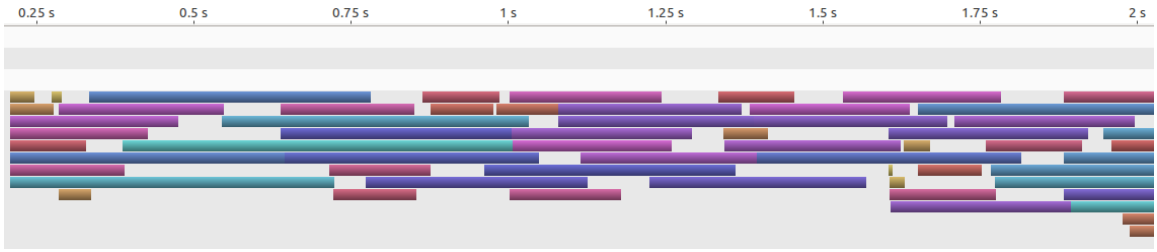

Figura 2: *Trace* de execução da ordenação $Standard$ com 64 kernels

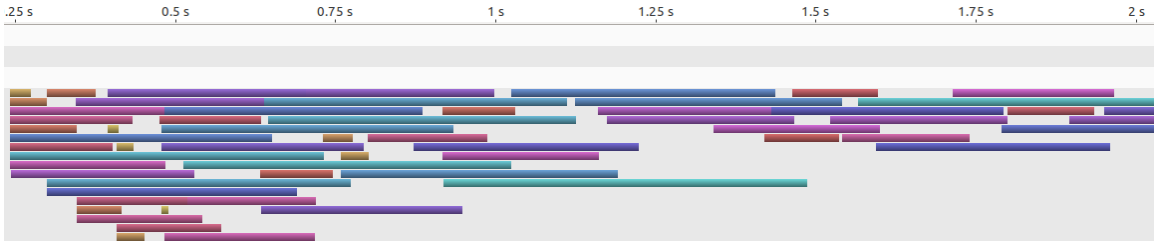

Figura 3: *Trace* de execução da ordenação $Greedy$ com 64 kernels

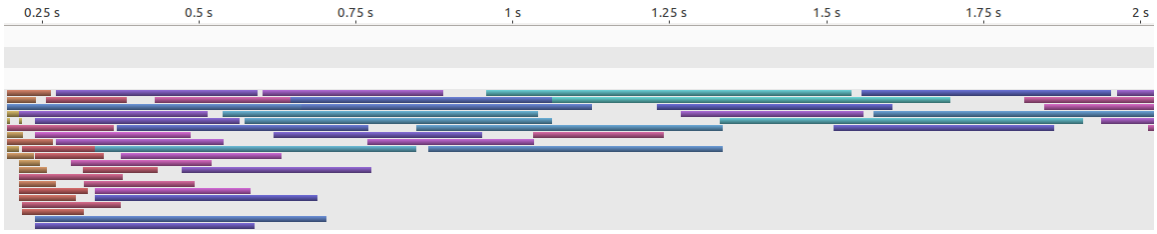

Figura 4: *Trace* de execução da ordenação $Dynamic$ com 64 kernels

As Figuras 5 e 6 mostram os resultados dos valores do $ANTT$ para diferentes configurações de execução, onde são alterados o número de *kernels*, N, e o número de blocos, NB. Nestes resultados, podemos observar que o método $Greedy$ fornece melhores resultados de ANTT em relação à ordenação $Standard$, os ganhos são de 17% a 70%. O método $Dynamic$ também fornece melhores resultados perante o $Standard$ de 55% a 86% de ganhos. Comparando $Greedy$ e $Dynamic$, observamos que $Dynamic$ fornece melhores resultados, porém os resultados de $Greedy$ ficaram mais próximos dos de $Dynamic$ do que dos de $Standard$.

As Figuras 7 e 8 mostram os valores do resultado do STP para as mesmas configurações dos resultados do $ANTT$, onde também são variados o número de *kernels*, N, e número de blocos, NB. Observamos ganhos parecidos em termos de throughput. $Greedy$ e $Dynamic$ aumentaram o throughput em relação à ordenação $Standard$. 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%. Porém os ganhos de $Dynamic$ em relação à $Greedy$ são menos pronunciados e menores com o aumento no número de kernels

6. Custo de Execução

O algoritmo de reordenação é realizado estaticamente antes dos *kernels* executarem. Cada reordenação possui um tempo de processamento e dependendo do seu tamanho pode representar um gargalo para a execução. Por exemplo, a implementação do problema da mochila com Programação Dinâmica requer a construção de uma matriz com uma dimensão considerável. Nos testes realizados neste trabalho, a diferença do tempo da reordenação $Greedy$ para o $Dynamic$ é significativa. O custo de execução de uma reordenação, chamado de *overhead*, representa a razão do

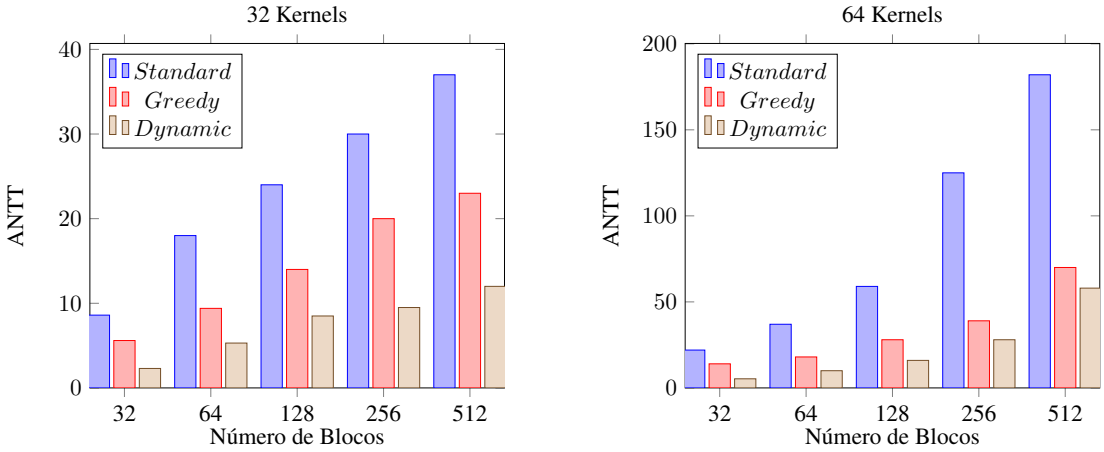

Figura 5: Resultado do ANTT para $N = 32$ e $N = 64$ na TitanX.

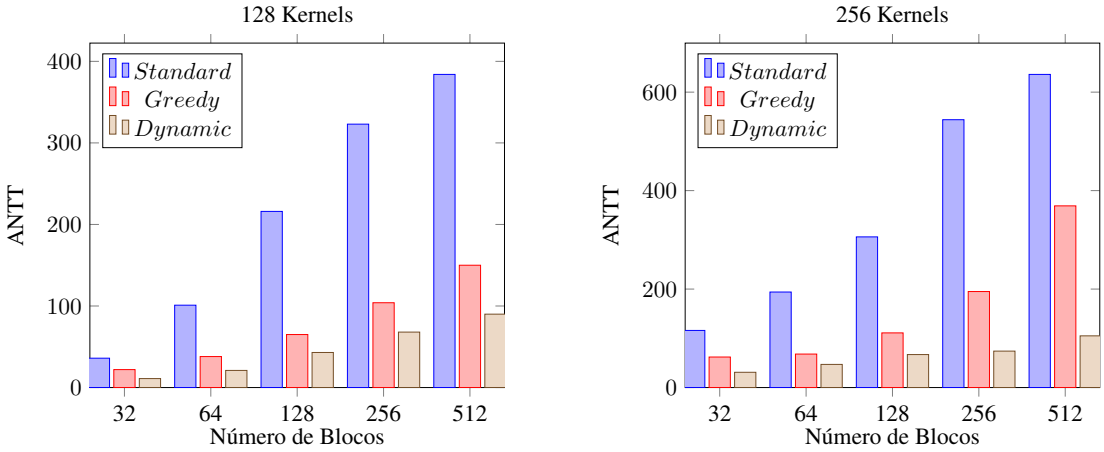

Figura 6: Resultado do ANTT para $N = 128$ e $N = 256$ na TitanX.

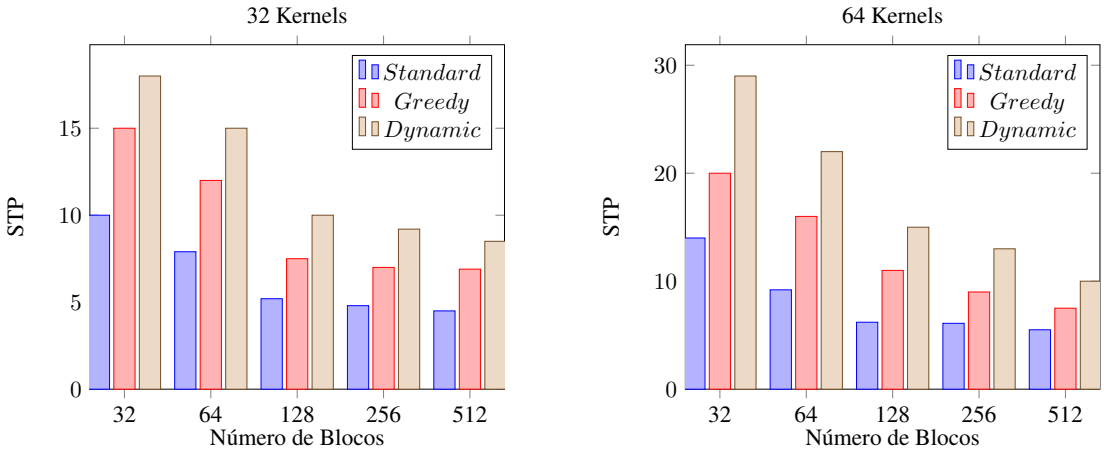

Figura 7: Resultado do STP para $N = 32$ e $N = 64$ na TitanX.

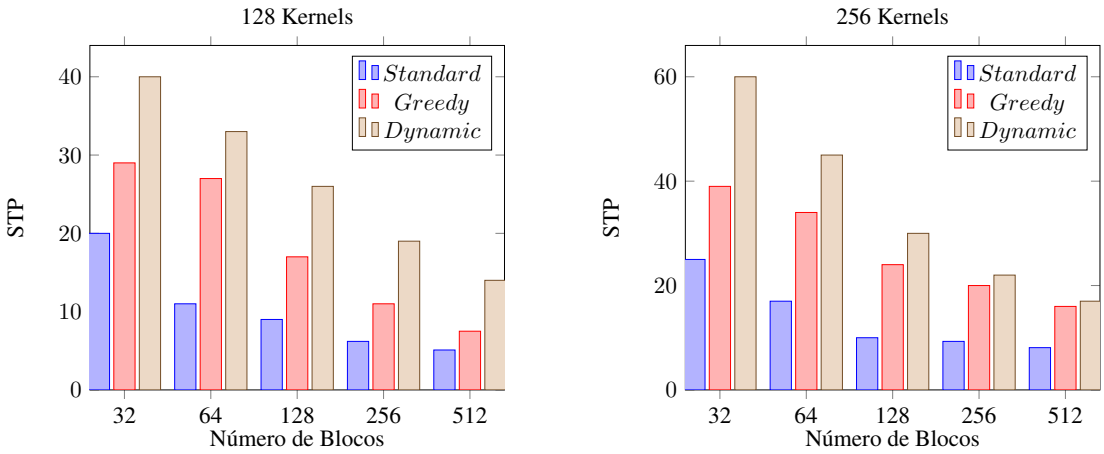

Figura 8: Resultado do STP para $N = 128$ e $N = 256$ na TitanX.

tempo de processamento com a diferença do *Average Turnaround Time* (ATT) entre a reordenação e o $Standard$. A equação 7 mostra a formula do ATT e a equação 8 mostra a fórmula do custo ou *overhead* O, onde OT representa o tempo de execução do algoritmo de ordenação.

$$ATT = \frac{\sum_{i=1}^{N} TT_i}{N} \tag{7}$$

$$O = \frac{OT \times ATT_{standard}}{(ATT_{standard} - ATT)} \tag{8}$$

As Tabelas 2 e 3 mostram os dois tipos de reordenação com suas respectivas diferença de *overhead*. Podemos observar que $O_{dynamic}$ é bem maior que O_{greedy}. O algoritmo de programação dinâmica é bem mais caro para executar. Quando o tamanho da matriz aumenta, o overhead aumenta também.

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 2: *Overhead* O_{greedy} do Guloso

$O_{dynamic}$	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 3: *Overhead* $O_{dynamic}$ da Programação Dinâmica

7. Conclusões

Este trabalho apresentou uma estratégia de reordenação dos *kernels* submetidos para executar na GPU. Utilizamos a proposta de trabalho anterior de modelar o problema como um problema da mochila 0-1 e propomos resolvê-lo utilizando um método simples, guloso, com um menor overhead.

Comparamos a abordagem gulosa com a abordagem de programação dinâmica proposta em [Breder et al. 2016]. 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. Conforme esperado, o método de programação dinâmica apresentou melhores resultados de ordenação. Porém, seu overhead de execução mostrou-se muito superior ao overhead gerado pelo método guloso. Concluímos que a heurística gulosa é uma alternativa viável para a reordenação de kernels, obtendo uma solução próxima da reordenação com programação dinâmica quando o número de *kernels* é muito grande.

Futuramente, pretendemos avaliar o uso de *benchmarks* da litaretura, com um estudo de caracterização dos *kernels* para avaliar o potencial de execução concorrente. Pretendemos também avaliar como a multiprogramação de *kernels* afeta o desempenho das aplicações.

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.

Breder, B., Charles, E., Cruz, R., Clua, E., Bentes, C., and Drummond, L. (2016). Maximizando o uso dos recursos de gpu atraves da reordenaçao da submissao de kernels concorrentes. In *XVII Simposio em Sistemas Computacionais de Alto Desempenho*, pages 98–109.

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

Duato, J., Pena, A. J., Silla, F., Mayo, R., and Quintana-Ortí, E. S. (2010). rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 224–231. IEEE.

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

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.

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