Soluções de alto desempenho para a Ordenação Segmentada de Vetores e o Escalonamento de Tarefas

Rafael Freitas Schmid

SERVIÇO DE PÓS-GRADUAÇÃO DA FACOM-UFMS

Data de Depósito:

Assinatura:______________________

Soluções de alto desempenho para a Ordenação Segmentada de Vetores e o Escalonamento de Tarefas[1]

Rafael Freitas Schmid

Orientador: *Prof. Dr. Edson Norberto Cáceres*

Tese apresentada à Faculdade de Computação - UFMS como parte dos requisitos necessários à obtenção do título de Doutor em Ciência da Computação.

UFMS - Campo Grande
Agosto/2021

[1]Trabalho Realizado com Auxílio da CAPES

Agradecimentos

Gostaria de agradecer, primeiramente, aos meus pais, que me deram a oportunidade de estudar, fazer uma faculdade e iniciar um mestrado, sem o qual eu talvez nem estaria aqui.

Agradeço também ao meus irmãos, que cresceram junto comigo e ajudaram a moldar a pessoa que sou hoje. Aos meus sogros e meu cunhado, que fizeram parte, bem de perto, de todo o processo do meu doutorado. Aos avós e tios da minha esposa, que me hospedaram e apoiaram durante todo o período que estive em Campinas.

Um agradecimento especial à minha esposa, que desde o mestrado me apoia nessa jornada de estudos e trabalho, sempre me incentivando e torcendo pelas minhas conquistas.

Ao Professor Dr. Edson Cáceres, meu orientador, um sincero agradecimento pela orientação e por todo aprendizado durante esses mais de quatro anos de estudos.

Agradeço ao Professor Dr. Edson Borin, pelas sugestões e orientações durante, e depois, do período em que estive na Universidade de Campinas (Unicamp). Ao Instituto de Computação da Unicamp e à equipe do LMCAD que, além das amizades construídas e o auxílio diário no desenvolvimento de minhas atividades, forneceram os recursos para a realização dos meus estudos no período em que estive lá.

Ao Instituto de Informática da Universidade Federal de Goiás (UFG) agradeço pela disponibilização do servidor onde os testes em múltiplas GPUs foram executados.

E, finalmente, um agradecimento aos apoios financeiros da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) e do Programa Nacional de Cooperação Acadêmica (Procad), que foram fundamentais para a realização de todo trabalho.

Abstract

Heterogeneous Computing Scheduling Problem (HCSP) consists of distribuiting the tasks between the available machines in the environment. Min-min heuristics have a good solution for this problem and can be very fast, compared with others for the same purpose. This work presents min-min implementations for the task scheduling problem in CPUs, and one implementation of the knapsack problem to reorder kernels in GPU. Before that, these two implementations were merged to solve the kernels scheduling problem in multiples GPUs. The most efficient algorithm for the min-min heuristic consists of solving the segmented sorting problem of an array. In this way, it was also realized a study about the existing segmented sorting implementations, and the new one suggested. The scheduling studies were realized in real development environments, and simulated environments of cloud computing, like the frameworks Cloudsim and GPUCloudsim. Results showed that heuristics, like min-min, can improve resource utilization on these tested environments.

Keywords: HCSP, Task Scheduling, Kernel Scheduling, Heterogeneous Computing, CloudSim, GPUCloudsim, GPU.

Resumo

O problema do escalonamento de tarefas consiste na distribuição de tarefas entre as máquinas disponíveis no ambiente. Uma das formas de resolver esse problema é usando a heurística min-min, que consegue resolvê-lo de forma rápida, ao mesmo tempo que entrega soluções eficientes. Neste trabalho, foram realizadas implementações da heurística min-min para o escalonamento de tarefas em múltiplas CPUs e uma implementação de mochila linear para o escalonamento de kernels em uma GPU. Além disso, essas duas estratégias foram unidas para resolver o escalonamento de kernels em múltiplas GPUs. O algoritmo mais eficiente da heurística min-min consiste em resolver o problema da ordenação segmentada de vetores. Dessa forma, foi realizado também um estudo sobre as implementações da ordenação segmentada existentes e comparadas com as novas sugeridas. Os estudos de escalonamento foram realizados em ambientes reais de desenvolvimento, bem como, em ambientes simulados de computação em nuvem, usando os frameworks Cloudsim e GPUCloudsim. Os resultados mostraram que o uso de heurísticas, como o min-min, pode otimizar a utilização dos recursos nesses ambientes.

Palavras-chave: Escalonamento de tarefas, CloudSim, GPUCloudsim, Reordenação de Kernel, Escalonamento de Kernel.

Conteúdo

Sumário xiii
Lista de Figuras xviii
Lista de Tabelas xxi
Lista de Abreviaturas xxiii
Lista de Algoritmos xxv

1 Introdução **1**
1.1 Motivação e Objetivos 2
1.2 Trabalho Desenvolvido 4
1.3 Principais Contribuições 4
1.4 Organização 5

2 Ordenação Segmentada de Vetores **7**
2.1 Trabalhos relacionados 8
2.2 Fix Sort 9
2.2.1 Estratégia Fix Padrão 9
2.2.2 Estratégia Fix Paralela 12
2.3 Análise dos Algoritmos 14
2.3.1 Radix Sort 14
2.3.2 Merge Sort 15
2.4 Resultados Experimentais 15
2.4.1 Comportamento com o Aumento no Número de Segmentos 17
2.4.2 Análise da Curva-S 20
2.4.3 Merge Sort Segmentado (**M**) vs. Hou Segmentado (**H**) 22
2.4.4 Fix Sort: CUB (**FC**) vs. Thrust (**FT**) 24
2.4.5 Guia para Escolha da Implementação 26
2.4.6 Tamanhos de Segmentos: Iguais vs. Diferentes 30
2.4.7 Impacto da Escolha de **FT** em cada Modelo de GPU 31
2.5 Conclusões e Trabalhos Futuros 32

3 Escalonamento de Tarefas em Ambiente de Computação Heterogênea 35
3.1 Trabalhos Relacionados 37
3.2 Soluções Propostas 41
3.2.1 Min-min clássico 42
3.2.2 Min-Min Sort 45
3.3 Resultados Experimentais 47
3.4 Conclusões e Trabalhos Futuros 52

4 Simulação do Escalonamento de Tarefas em um Ambiente de Computação em Nuvem 55
4.1 Ambiente em Nuvem 56
4.1.1 CloudSim 57
4.1.2 GPUCloudsim 58
4.2 Simulação do Escalonamento de Tarefas usando o Cloudsim . . . 62
4.2.1 Experimentos e Estratégias de Escalonamento 63
4.2.2 Resultados e Análises 66
4.3 Simulação do Escalonamento de Kernels em GPUs usando o GPU-Cloudsim 71
4.3.1 Ambiente com uma única GPU 72
4.3.2 Ambiente com Múltiplas GPUs 73
4.4 Conclusões e Trabalhos Futuros 78

5 Reordenação de Kernels em GPUs 81
5.1 Arquitetura de uma GPU 82
5.2 Maximizando a Ocupação da GPU 84
5.3 Reordenação de Kernels 86
5.4 Implementações 87
5.5 Resultados Experimentais 90
5.6 Análise das Estratégias em Relação ao Número de Filas 92
5.6.1 Análise da Estratégia Padrão (**P**) 92
5.6.2 Análise da Estratégia Fit (**F**) 98
5.6.3 Análise da Estratégia da Mochila Unidimensional (**ML**) . . . 103
5.7 Comparação entre as Estratégias 106
5.8 Análise das Filas de Execução 109
5.9 Ambiente com múltiplas GPUs 114
5.10 Conclusões e Trabalhos Futuros 117

6 Conclusões e Trabalhos Futuros 121
6.1 Resumo dos Objetivos e Principais Resultados 122
6.2 Limitações 124

6.3 Trabalhos Futuros 124
6.4 Publicações Derivadas do Trabalho 125

Referências **132**

A Apêndice **133**

Lista de Figuras

2.1 Tempo de execução das estratégias com o aumento no número de segmentos em vetores com 2^{25} elementos na Quadro 4000. . . 17
2.2 Curva-S das estratégias considerando os resultados de todas as GPUs. 21
2.3 Comportamento da estratégia **H** considerando diferentes números de segmentos e tamanhos de vetores na GTX 1070. 23
2.4 *Speedup* da estratégia **FT** quando comparada à estratégia **FC** considerando somente o passo de ajuste (*fix*), somente o passo de ordenação (*sort*), e todo o processo *fix sort*. 25
2.5 Curva-S dos cenários nos quais **FC** é a estratégia com o menor impacto entre todas as GPUs. 29
2.6 Curva-S dos cenários nos quais **M** é a estratégia com o menor impacto entre todas as GPUs. 29
2.7 Curva-S dos cenários nos quais **FT** é a estratégia com o menor impacto entre todas as GPUs. 29
2.8 Curva-S dos cenários nos quais **MT** é a estratégia com o menor impacto entre todas as GPUs. 29
2.9 Curva-S dos cenários nos quais **R** é a estratégia com o menor impacto entre todas as GPUs. 29
2.10 Curva-S dos cenários nos quais **H** é a estratégia com o menor impacto entre todas as GPUs. 29

3.1 Estratégia de implementação em GPU da heurística min-min (Nesmachnow e Canabé, 2011). 44
3.2 Comportamento das implementações ao aumentar o número de máquinas, enquanto mantém a quantidade de tarefas em 32768. 48
3.3 Comportamento das implementações ao aumentar o número de tarefas, enquanto mantém a quantidade de máquinas em 512. . . 49
3.4 Tempo de execução das implementações do *min-min sort*. 49

3.5 *Speedup* das implementações do min-min em relação à implementação sequencial padrão. 50

4.1 Relação entre as entidades de um ambiente de Computação em Nuvem. 57
4.2 Funcionamento do CloudSim (Li et al., 2011). 59
4.3 (a) Principais componentes de um servidor equipado com GPU e (b) as VMs que suportam GPUs (**?**). 60
4.4 (a) Placa de vídeo e os recursos das GPUs. (b) Os passos de execução de uma tarefa na GPU (**?**). 61
4.5 Média do Tempo de Execução de Cada Heurística para as Dimensões do Problema. 68
4.6 Quantidade de Tarefas em Cada VM. 69
4.7 Tempo de Execução de Cada VM para uma das Entradas de Dimensão 16 máquinas e 1024 tarefas. 70
4.8 *Speedup* da implementação **MT** em relação ao escalonamento padrão **P**. 73
4.9 *Speedup* da implementação **MinMT** em relação às outras implementações em um **ambiente homogêneo** com: (a) 2 VMs, (b) 4 VMs e (c) 8 VMs. 76
4.10 *Speedup* da implementação **MinMT** em relação às outras implementações em um **ambiente heterogêneo** com: (a) 2 VMs, (b) 4 VMs e (c) 8 VMs. 78

5.1 Diferença entre a arquitetura de uma CPU e de uma GPU (**?**). . . 82
5.2 Os blocos de *threads* podem ter até 3 dimensões e são distribuídos entre os SMs da GPU. 83
5.3 A concorrência da Fermi é limitada a uma única fila de trabalho. Na Kepler, as *streams* podem executar concorrentemente usando filas separadas de tarefas (Breder, 2016). 86
5.4 *Speedup* do tempo de execução paralelo em relação ao tempo de execução sequencial para 256 *kernels*, usando a estratégia padrão. 93
5.5 Proporção de melhora da média do tempo de resposta da execução paralela em relação ao tempo de resposta da execução sequencial para 256 *kernels*, usando a estratégia padrão. 94
5.6 Impacto no tempo de duração da execução paralela em relação à execução sequencial para 256 *kernels*, usando a estratégia padrão. 95
5.7 Impacto na média dos watts da execução paralela em relação à execução sequencial para 256 *kernels*, usando a estratégia padrão. 96

5.8 Impacto da quantidade de joules consumido durante a execução paralela em relação à execução sequencial para 256 *kernels*, usando a estratégia padrão. 97
5.9 Execução paralela dos *kernels* usando a estratégia Padrão (**P**). . . 97
5.10 *Speedup* do tempo de execução paralelo em relação ao tempo de execução sequencial para 256 *kernels*, usando a estratégia fit. . . 98
5.11 Proporção de melhora da média do tempo de resposta da execução paralela em relação ao tempo de resposta da execução sequencial para 256 *kernels*, usando a estratégia fit. 99
5.12 Impacto no tempo de duração da execução paralela em relação à execução sequencial para 256 *kernels*, usando a estratégia fit. . . 100
5.13 Impacto na média do watts da execução paralela em relação à execução sequencial para 256 *kernels*, usando a estratégia fit. . . 101
5.14 Impacto na quantidade de joules consumido durante a execução paralela em relação à execução sequencial para 256 *kernels*, usando a estratégia fit. 102
5.15 Execução paralela dos *kernels* usando a estratégia fit (**F**). 102
5.16 *Speedup* do tempo de execução paralelo em relação ao tempo de execução sequencial para 256 *kernels*, usando a estratégia da mochila. 103
5.17 Proporção de melhora da média do tempo de resposta da execução paralela em relação ao tempo de resposta da execução sequencial para 256 *kernels*, usando a estratégia da mochila. 104
5.18 Impacto no tempo de duração da execução paralela em relação à execução sequencial para 256 *kernels*, usando a estratégia da mochila. 105
5.19 Impacto na média dos watts da execução paralela em relação à execução sequencial para 256 *kernels*, usando a estratégia fit. . . 106
5.20 Impacto na quantidade de joules consumido durante a execução paralela em relação à execução sequencial para 256 *kernels*, usando a estratégia da mochila. 107
5.21 Execução paralela dos *kernels* usando a estratégia da Mochila (**ML**). 107
5.22 *Speedup* do tempo de execução paralelo em relação ao tempo de execução sequencial para 256 *kernels*. 108
5.23 Proporção de melhora da média do tempo de resposta da execução paralela em relação ao tempo de resposta da execução sequencial para 256 *kernels*. 109
5.24 Impacto no tempo de duração da execução paralela em relação à execução sequencial para 256 *kernels*. 110

5.25 Impacto na média dos watts da execução paralela em relação à execução sequencial para 256 *kernels*. 111
5.26 Impacto no número de joules consumido durante a execução paralela em relação à execução sequencial para 256 *kernels*. 112
5.27 O primeiro kernel possui 104 blocos com 256 threads cada um. O limite do número de threads é atingido pelos SMs e nenhum outro kernel pode iniciar até que ele termine. 113
5.28 O primeiro kernel possui 103 blocos com 256 threads cada um. O limite do número de threads por SM é atingido em 12 SMs, mas um deles ainda tem disponível 256 threads, que é o número de threads do segundo kernel. 113
5.29 Todos os *kernels* cabem juntos na GPU. 114
5.30 Kernels grandes sendo executados em *streams* diferentes. 114
5.31 Apenas o kernel 11 não cabe na GPU junto com os outros. 115
5.32 Um exemplo ideal, quando os *kernels* das três filas de execução cabem juntos na GPU. 115
5.33 Um exemplo da mudança da ordem execução, quando um kernel de uma fila finaliza antes que um kernel que foi despachado primeiro. 116
5.34 *Speedup* das implementações em relação à estratégia padrão, que distribui os *kernels* igualmente entre as GPUs. 116
5.35 *Speedup* das implementações em relação à estratégia padrão, que distribui os *kernels* igualmente entre as GPUs. 117
5.36 *Speedup* das implementações em relação à estratégia padrão, que distribui os *kernels* igualmente entre as GPUs. 118

Lista de Tabelas

2.1 Exemplo do pré-processamento da estratégia *fix* padrão. 10
2.2 Exemplo do pós-processamento da estratégia *fix* padrão. 11
2.3 Exemplo da limitação do pré-processamento para um número de 8 *bits*. 11
2.4 Exemplo do preprocessamento *fix sort* paralelo. 13
2.5 Especificações de hardware e configurações de software de cada máquina. 16
2.6 Especificações das GPUs. 16
2.7 Exemplo da estratégia *fix sort* ao ordenar um vetor com 8 elementos e 2 segmentos. 19
2.8 Exemplo da estratégia *fix sort* ao ordenar um vetor com 8 elementos e 4 segmentos. 19
2.9 Percentual de casos onde cada estratégia está dentro do *slowdown* máximo especificado quando comparadas com a estratégia mais rápida. 21
2.10 Percentual de vezes em que a estratégia **FT** é a mais rápida em cada combinação de tamanho do vetor e número de segmentos, considerando todas as GPUs. 27
2.11 Percentual de vezes em que a estratégia **FC** é a mais rápida em cada combinação de tamanho do vetor e número de segmentos, considerando todas as GPUs. 27
2.12 Percentual de vezes em que a estratégia **H** é a mais rápida em cada combinação de tamanho do vetor e número de segmentos, considerando todas as GPUs. 27
2.13 Percentual de vezes em que a estratégia **M** é a mais rápida em cada combinação de tamanho do vetor e número de segmentos, considerando todas as GPUs. 27

2.14 Percentual de vezes em que a estratégia **R** é a mais rápida em cada combinação de tamanho do vetor e número de segmentos, considerando todas as GPUs. 27
2.15 Percentual de vezes em que a estratégia **MT** é a mais rápida em cada combinação de tamanho do vetor e número de segmentos, considerando todas as GPUs. 27
2.16 Cada cenário com a estratégia que resulta na escolha que gera o menor impacto considerando os slowdowns das estratégias em todas as GPUs. 28
2.17 Percentual de casos em que cada estratégia está dentro do *slowdown* máximo especificado, ao seguir as recomendações da Tabela 2.16. 28
2.18 Percentual de casos onde cada estratégia está dentro do *slowdown* máximo especificado, quando comparada com a estratégia mais rápida considerando somente segmentos do mesmo tamanho. 30
2.19 Percentual de casos onde cada estratégia está dentro do *slowdown* máximo especificado, quando comparada com a estratégia mais rápida considerando somente segmentos do mesmo tamanho. 31
2.20 Dados estatísticos com o impacto de sempre escolher o *fix sort* para realizar a ordenação segmentada. 32

3.1 Exemplo de execução do algoritmo *min-min clássico* em um ambiente com 3 máquinas e 5 tarefas. 44
3.2 Exemplo do algoritmo *min-min sort* em um ambiente com 3 máquinas e 5 tarefas. 46
3.3 Tempos de execução das implementações do *min-min clássico* e os respectivos *speedups* quando comparadas com a versão sequencial. 50
3.4 Tempos de execução das implementações do *min-min sort* e o *speedup* das mesmas, quando comparadas com a versão sequencial. 51
3.5 Melhor implementação para cada dimensão do problema rodando na Quadro M4000. 52

4.1 Média e o Desvio Padrão do Número de Instruções Gerados nos Arquivos de Entrada. 64
4.2 Cenário 1: Relação entre Tarefas/Máquinas igual em todos os testes. 65
4.3 Cenário 2: Poder computacional do ambiente e quantidade de tarefas iguais em todos os testes. 66
4.4 Média e desvio padrão dos *makespans* para cada dimensão do problema. 66

4.5 *Speedup* em relação ao algoritmo padrão do Cloudsim. 67
4.6 Tempo de Escalonamento das Heurísticas para as Dimensões do Problema 67
4.7 Média e Desvio Padrão dos *Makespans* para cada Dimensão. . . . 68
4.8 Percentual de Utilização dos Datacenters e Preço Hipotético Sugerido 70
4.9 Tempos de execução em uma GPU. 72
4.10 Tempos de execução em GPUs com configurações iguais. 75
4.11 Frequência de execução das GPUs no ambiente heterogêneo. . . . 77
4.12 Tempos de execução em GPUs com configurações diferentes. . . . 77

5.1 Ordem de despacho dos *kernels* efetuada pelo usuário. Referente à Figura 5.32 114

A.1 Tempo de execução das estratégias para cada dimensão do problema. 133
A.1 Tempo de execução das estratégias para cada dimensão do problema. 134
A.1 Tempo de execução das estratégias para cada dimensão do problema. 135
A.1 Tempo de execução das estratégias para cada dimensão do problema. 136
A.1 Tempo de execução das estratégias para cada dimensão do problema. 137
A.1 Tempo de execução das estratégias para cada dimensão do problema. 138
A.1 Tempo de execução das estratégias para cada dimensão do problema. 139
A.1 Tempo de execução das estratégias para cada dimensão do problema. 140

Lista de Abreviaturas

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

ETC Expected Time to Compute

HC Heterogeneous Computing

FC Implementação do Fix Sort usando a biblioteca CUB

FiFo First In First Out

FT Implementação do Fix Sort usando a biblioteca Thrust

G Implementação da reordenação de kernel usando o algoritmo guloso

GA Genetic Algorithm

GSA Genetic Simulated Annealing

GPU Graphic Processing Unit

HCSP Heterogeneous Computing Scheduling Problem

Hou Implementação da ordenação segmentada de Hou et al.

LSD Least Significant Digit

M Implementação da ordenação segmentada da biblioteca ModernGPU

MCT Minimum Completion Time

MET Minimum Execution Time

Min Implementação da distribuição de kernels entre GPUs usando a heurística min-min

MinMT Implementação da distribuição de kernels entre GPUs usando a heurística min-min e depois a reordenação de kernels em cada GPU usando uma mochila

ML Implementação da reordenação de kernel usando o algoritmo da mochila com linearização do peso

MSD Most Significant Digit

MSB Most Significant Bit

MT Implementação da reordenação de kernel usando o algoritmo da mochila e o número de threads como peso

OLB Opportunistic Load Balancing

P Implementação da reordenação de kernel distribuindo igualmente entre os streams

R Implementação da ordenação segmentada da biblioteca CUB

SA Simulated Annealing

SIMT Single Instruction, Multiple Thread

SLA Service Level Agreement

SM Streaming Multiprocessor

VM Virtual Machine

Lista de Algoritmos

1 Pseudo-código do pré-processamento (estratégia fix padrão). . . . 10
2 Pseudo-código do pós-processamento (estratégia fix padrão). . . . 10
3 Pseudocódigo para o pré-processamento do vetor (estratégia *fix* paralela). 12
4 Pseudocódigo para o pós-processamento do vetor (estratégia *fix* paralela). 13
5 Pseudocódigo do pré-processamento de um vetor realizado pela estratégia *fix* paralela. 13
6 Pseudocódigo do pós-processamento de um vetor realizado pela estratégia *fix* paralela. 14
7 min-min clássico 42
8 Min-Min Sort 45
9 Pseudocódigo da implementação de escalonamento padrão (**P**). . . 88
10 Pseudocódigo das implementações de escalonamento: **G**, **F** e **ML**. 89
11 Pseudocódigo da seleção de *kernels* fit (**F**). 89
12 Pseudocódigo da seleção de *kernels* usando o algoritmo da mochila (**ML** e **MT**). 90

CAPÍTULO 1

Introdução

Neste trabalho propomos e avaliamos algoritmos para os problemas de ordenação segmentada, escalonamento de tarefas em um ambiente de computação heterogênea e reordenação de *kernels* em GPUs.

A ordenação segmentada consiste na ordenação de linhas de uma matriz ou segmentos de um vetor e é utilizada na solução de problemas tais como: um método de regressão para diagnóstico em tempo real de plasma (Ferreira et al., 2015), algoritmo de reclassificação de imagens (Pisani et al., 2015), a construção de vetor de sufixo de uma *string* (Wang et al., 2015) e em algoritmos para escalonamento de tarefas.

Em computação de Alto Desempenho, um dos desafios é manter a taxa de ocupação dos recursos computacionais a mais alta possível, seja em um ambiente heterogêneo ou num ambiente homogêneo.

O problema de escalonamento de tarefas em ambiente de computação heterogênea (HCSP - Heterogeneous Computing Scheduling Problem) consiste na distribuição das tarefas a serem executadas nas máquinas disponíveis, sendo que essas máquinas podem ter configurações diferentes e desempenhos diferentes ao executar cada uma das tarefas. O HCSP é utilizado em computação distribuída, no escalonamento de tarefas em ambientes heterogêneos, tais como grids e computação em nuvem.

No caso de um ambiente homogêneo, um problema a ser resolvido é o de garantir que um dispositivo *multicore* utilize de forma efetiva todos os seus núcleos de processamento. Uma das formas de otimizar essa utilização pode ser feita por meio de um escalonamento dos *kernels* , pois os atuais dispositivos *multicore* estão equipados com tecnologias que permitem que dois ou mais *kernels* executem ao mesmo tempo em uma mesma GPU, quando houver re-

cursos disponíveis para isso (Breder, 2016). Dessa forma, a ordem em que os *kernels* são escalonados para execução pode interferir no tempo de execução final, bem como os *kernels* selecionados para executar em cada uma das GPUs disponíveis impacta diretamente no desempenho dos programas em execução.

No problema da ordenação segmentada, estudamos as implementações existentes em GPU, e propusemos um novo algoritmo específico, além disso, elaboramos um *benchmark* para avaliar o desempenho dos algoritmos existentes em diferentes configurações. O novo algoritmo proposto mostrou-se eficiente e competitivo, inclusive quando comparado com implementações sequenciais em CPU.

O HCSP é um problema NP-completo (Ullman, 1975), e por isso, muitas heurísticas têm sido aplicadas para resolvê-lo, como algoritmos genéticos (GA), *Genetic Simulated Annealing* (GSA), *Suffrage*, *min-min*, *max-min*, etc (Braun et al., 2001). Em função de suas características, a heurística min-min (Ezzatti et al., 2013) é uma das mais rápidas e eficientes para esse problema e obtém soluções com qualidade comparáveis às obtidas com GA, e com tempo de execução bem menor (Min-You Wu et al., 2000). Usando o benchmark desenvolvido para ordenação segmentada, propomos duas soluções para o problema do HCSP usando a heurística min-min, sendo uma em GPU e outra híbrida (CPU/GPU). As duas implementações também foram avaliadas em um ambiente simulado de computação em nuvem, no qual se comparou a heurística com o escalonador nativo do framework Cloudsim. Os testes indicam que as soluções propostas são competitivas.

Na reordenação de *kernels* em GPUs, estudamos três estratégias: a mochila unidimensional, a heurística min-min e uma implementação híbrida em que combinamos a mochila unidimensional com as heurísticas min-min e max-min para escalonar as tarefas. Para uma única GPU avaliamos a estratégia da mochila unidimensional, na qual a ordem de execução dos *kernels* é definida a cada interação por meio da solução de uma mochila unidimensional que considera o número de threads e o tempo de execução estimado. Usando uma GPU, um ambiente com 4 GPUs e o GPUCloudsim (**?**) (ambiente simulado para computação em nuvem com múltiplas GPUs), avaliamos o comportamento das estratégias de reordenação e escalonamento de *kernels* nesses ambientes computacionais.

1.1 Motivação e Objetivos

A solução eficiente do problema de ordenação segmentada impacta de forma positiva a solução de vários outros problemas. Dentre esses problemas, destacamos a melhoria no desempenho de algoritmos de escalonamento de tarefas

em ambientes heterogêneos e ambientes *multicores*. O *fix sort* Schmid e Cáceres (2019) foi um algoritmo apresentado nesse trabalho que faz um ajuste nos elementos do vetor para permitir a utilização de uma implementação de ordenação de único segmento, quando comparado com outras implementações, apresentou-se como o melhor dependendo da configuração das dimensões do problema. Nesse caso, a escolha do algoritmo mais indicado para a configuração do problema a ser resolvido tem um grande impacto no tempo de execução desse problema. Isso motivou um estudo detalhado e a realização de um benchmark com a análise dos comportamentos de vários algoritmos para várias configurações de entrada e execução. Os resultados obtidos com *fix sort* e com o *benchmark* possibilitaram novas abordagens para a implementação da heurística min-min que foi utilizada na solução do problema do escalonamento de tarefas em ambiente heterogêneo (HCSP) e do problema de reordenação de *kernels* em múltiplas GPUs.

Como o HCSP é um problema NP-Completo, várias heurísticas foram propostas para solucioná-lo. A heurística min-min, além de obter boas soluções para esse problema, é uma das mais rápidas. Dessa forma, encontrar meios de otimizar essa heurística trará impactos positivos para a solução desse problema. Uma das formas de implementar a heurística min-min é com a utilização da ordenação segmentada. Usando o estudo do comportamento da ordenação segmentada projetamos dois algoritmos para resolver o HCSP, sendo um em GPUs, e outro em CPU/GPU. Esses algoritmos foram adaptados para computação em nuvem e implementados no simulador CloudSim.

No problema do escalonamento de tarefas em ambiente de computação heterogênea, várias tarefas estão disponíveis para execução e devem ser escalonadas em máquinas com capacidade e características diferentes, o que viabiliza a utilização da heurística min-min. Já no problema de reordenação de *kernels* em uma única GPU, existem várias filas de *kernels* para serem submetidos na mesma GPU, que serão executados em paralelo apenas se houver recursos disponíveis nela. Nesse caso, a aplicação da heurística min-min não pode ser feita diretamente. Porém, quando estendemos o problema para reordenação de *kernels* em múltiplas GPUs, ele pode ser visto como um problema de escalonamento de tarefas em computação heterogênea, já que existem diversos *kernels* , ou tarefas, a serem escalonados em diferentes GPUs, que podem ter poder computacional diferentes. Para o caso de uma única GPU, avaliamos a utilização de uma mochila unidimensional para fazer a escolha dos *kernels* a serem executados a cada interação de acordo com o número de threads de cada kernel.

No nosso trabalho propomos novas formas eficientes de resolver os problemas de ordenação segmentada, escalonamento de tarefas em ambientes

heterogêneos e reordenação de *kernels* em GPUS.

1.2 Trabalho Desenvolvido

Neste trabalho realizamos um estudo sobre implementações eficientes do problema de ordenação segmentada. Além disso, projetamos e implementamos um algoritmo competitivo, denominado *fix sort*, que reajusta o vetor de tal forma que uma implementação tradicional de ordenação possa ser invocada para resolver o problema. O algoritmo proposto, funciona para segmentos de tamanhos fixos ou variados e pode ser usado em ambiente sequencial ou paralelo. Além disso, um *benchmark* com as melhores implementações para cada dimensão do problema foi elaborado e serve como um guia para os desenvolvedores que necessitem usar ordenação segmentada.

Esses estudos auxiliaram no desenvolvimento da implementação híbrida (CPU/GPU) de um algoritmo linear da heurística min-min para o HCSP, usando um algoritmo de ordenação segmentada baseado no radix sort, que se mostrou competitivo com a implementação baseada no merge sort de Ezzatti et al. (2013). Posteriormente o algoritmo também foi implementado em uma GPU. Essa implementação também foi utilizada no escalonamento de tarefas em um ambiente de computação em nuvem, usando o Cloudsim, no qual a implementação da heurística min-min foi comparada com o algoritmo de escalonamento nativo da ferramenta, baseado no FCFS (First Come First Served).

Um outro problema estudado foi da reordenação de *kernels* em uma única GPU. Utilizamos o problema da mochila unidimensional para escalonamento das filas de tarefas e avaliamos o comportamento do escalonamento de tarefas na GPU, utilizando várias métricas, tais como o tempo de execução e o consumo de energia para várias configurações. Comparamos os resultados obtidos com o trabalho de Breder (2016). Esse problema foi estendido para um ambiente de computação com múltiplas GPUs, no qual identificamos que a heurística min-min poderia ser empregada na distribuição das tarefas entre as máquinas e auxiliar na otimização da solução. Os testes desses algoritmos também foram realizados em um ambiente simulado de computação em nuvem, com a utilização da ferramenta GPUCloudsim, que é uma extensão do Cloudsim, mas com suporte para GPUs.

1.3 Principais Contribuições

As principais contribuições deste trabalho consistiram na apresentação de uma implementação competitiva de ordenação segmentada em GPU, de um guia para os desenvolvedores com implementações eficientes para o problema

da ordenação segmentada de vetores, uma implementação híbrida (CPU/GPU), linear e eficiente da heurística min-min, uma proposta de escalonador para o Cloudsim, uma estratégia de escalonamento de *kernels* em uma única GPU, utilizando mochila unidimensional, e a apresentação de estratégias de escalonamento de *kernels* em múltiplas GPUs em um ambiente de nuvem.

1.4 Organização

O trabalho está organizado da seguinte maneira. O Capítulo 2 descreve os estudos sobre implementações em GPU do problema de ordenação segmentada. O Capítulo 3 apresenta os fundamentos do HCSP e as heurísticas mais conhecidas para resolvê-lo. Além disso, são apresentadas estratégias de implementação da heurística min-min em CPU e GPU e os resultados obtidos com essas implementações. Na sequência, o Capítulo 4 apresenta os resultados dos algoritmos do capítulo anterior sendo executados em um ambiente em nuvem simulado, usando o CloudSim, além de implementações de escalonamento de *kernels* em ambiente simulado usando o GPUCloudSim. A formalização do problema de reordenação de *kernels* , as características da GPU que permitem a execução paralela entre *kernels* , e os resultados obtidos com os algoritmos implementados, são descritos no Capítulo 5. As conclusões e trabalhos futuros são apresentadas no Capítulo 6.

CAPÍTULO 2

Ordenação Segmentada de Vetores

Neste capítulo, comparamos as implementações CUDA existentes para a ordenação segmentada, realizando *benchmarks* de desempenho personalizados, com o objetivo de identificar e entender a estratégia que possui melhor desempenho para cada caso. Uma vez que pretendemos que estes resultados sejam utilizados por desenvolvedores, apresentamos um guia simples que pode ser seguido quando o problema possui características similares às dos nossos testes.

A principais contribuições deste trabalho são:

1. Uma comparação das implementações mais rápidas da ordenação segmentada de vetores em GPU (Hou et al., 2017; Schmid e Cáceres, 2019) em sete modelos de GPUs diferentes, em uma variedade de cenários, incluindo cenários com diferentes número de segmentos e considerando segmentos do mesmo tamanho e de tamanhos diferentes.

2. Análise assintótica dos algoritmos para explicar como o número de segmentos afeta o desempenho de cada implementação.

3. Apresentação da estratégia *fix sort*, que apresentou resultados eficientes, sendo a melhor opção em 44,11% dos casos.

4. Um mapa que indica a melhor implementação para cada dimensão do problema.

5. Análise da curva-s para avaliar o desempenho das implementações recomendadas.

6. Uma análise da implementação **FT** considerando diferentes modelos de GPU e segmentos de tamanhos iguais e diferentes.

O restante do capítulo é organizado como se segue: Na Seção 2.1 avaliamos os estudos relacionados à ordenação segmentada, seguido pela Seção 2.2, que detalha o algoritmo **fix sort**. Uma análise dos algoritmos utilizados nas implementações é realizado na Seção 2.3.Na Seção 2.4 apresentamos a análise dos resultados experimentais, e na Seção 2.5 as conclusões, bem como uma lista de contribuições e possibilidade de trabalhos futuros.

2.1 Trabalhos relacionados

Nesta seção, apresentamos alguns dos principais trabalhos relacionados à implementação de ordenação segmentada em GPUs.

Bergstrom e Reppy (2012) descreveram uma forma para permitir que a linguagem NESL [1] possa ser executada em GPUs e compara algumas operações em seus *benchmarks*. Nesse trabalho, a operação de ordenação segmentada faz múltiplas chamadas ao *radix sort* da biblioteca Thrust, uma vez para cada segmento. Porém, dependendo do número de segmentos, o *overhead* associado às múltiplas chamadas pode dominar o tempo de execução, conforme descrito por Schmid et al. (2016).

Baxter (2013) propôs um algoritmo baseado no *merge sort* para ordenar múltiplos segmentos de um vetor, denominado *segmented sort*, e sua implementação CUDA é parte da biblioteca MGPU, também escrita por ele. Não temos conhecimento de publicações sobre esta implementação, mas a página da biblioteca apresenta o desempenho para várias entradas diferentes. Neste trabalho, vamos nos referir a essa implementação como *segmented merge sort* (**M**).

A biblioteca CUB é um projeto de código aberto desenvolvido pela NVIDIA Research Corporation (NVIDIA, 2016). Ela inclui a função *segmented radix sort*, que é capaz de ordenar múltiplos segmentos de um vetor. O algoritmo utiliza uma versão paralela do *radix sort* para cada segmento do vetor. Apesar de obter bons desempenhos em algumas situações, há entradas em que este método deixa de ser eficiente, devido à má configuração dos blocos CUDA.

Schmid et al. (2016) descreveram uma estratégia para executar a ordenação segmentada, chamada *fix sort* (**FT** e **FC**). Esta estratégia consiste em executar um pré e um pós processamento no vetor, para permitir que um algoritmo paralelo padrão de ordenação seja utilizado para ordenar os segmentos. Posteriormente, Schmid e Cáceres (2019) apresentaram uma análise

[1] Linguagem funcional de primeira ordem projetada para permitir que os programadores escrevam programas paralelos irregulares em processadores vetorizados.

comparando o desempenho da ordenação segmentada em CPU e GPU, tendo a GPU obtido os melhores resultados para resolver essa tarefa.

Um algoritmo de ordenação segmentada foi implementado por Hou et al. (2017) e está disponível no Github[2]. Esta implementação baseia-se no merge sort e funciona de modo similar à implementação do Modern GPU (**M**). Neste trabalho, essa implementação será referenciada como *hou segmentado* (**H**).

Todas as estratégias descritas nesta seção foram comparadas para várias dimensões de vetores e tamanhos de segmentos. Nosso *benchmark* compara dois tipos de vetores: (1) segmentos de tamanhos diferentes e (2) segmentos de tamanhos iguais.

2.2 Fix Sort

Como discutido anteriormente, invocar múltiplas vezes um algoritmo de ordenação tradicional, um para cada segmento, pode causar um grande *overhead* de desempenho. Com o objetivo de testar esses algoritmos tradicionais de ordenação para ordenar múltiplos segmentos com uma única chamada, propusemos um método que ajusta os valores armazenados em cada segmento e garante que ao ordenar o vetor inteiro, composto pela concatenação de todos os segmentos, não movemos elementos de um segmento para outro. Denominamos esta estratégia de *fix sort*.

A ideia chave por trás do *fix sort* é adicionar um valor constante (fator *fix*) aos elementos de cada segmento, garantindo que seus valores tornem-se maiores ou iguais aos valores armazenados nos segmentos à sua esquerda. Uma vez que os elementos são ajustados, um algoritmo de ordenação tradicional ordena o vetor. Então, os valores de cada elemento do vetor são restaurados, subtraindo o fator *fix*.

Nesta seção, apresentamos duas implementações diferentes dessa estratégia. Uma é mais flexível e pode lidar com um grande número de segmentos, enquanto a outra é altamente paralelizável.

2.2.1 Estratégia Fix Padrão

A primeira versão do *fix sort* consiste em somar aos elementos de cada segmento um fator constante. Esse fator é calculado subtraindo o maior valor do segmento anterior do menor valor do segmento corrente. O Algoritmo 1 mostra o pseudo-código do pré-processamento, em que arr representa o vetor de dados e o seg representa o vetor de índices de segmentos. Os segmentos são ajustados um a um e a variável $prevMax$ armazena o maior valor encontrado no segmento anterior.

[2] https://github.com/vtsynergy/bb_segsort

Algoritmo 1 Pseudo-código do pré-processamento (estratégia fix padrão).

```
1: procedure PREPROCESS(arr, seg)
2:     prevMax ← 0
3:     for i = 0, size(seg)-1 do
4:         currMin ← smallest element of segment i
5:         fix(i) ← prevMax - currMin
6:         for j = seg(i), seg(i+1)-1 do
7:             arr(j) ← arr(j) + fix(i)
8:         prevMax ← largest element of segment i
```

O Algoritmo 2 mostra o pseudo-código do pós-processamento. Neste caso, a computação pode ser feita completamente em paralelo.

Algoritmo 2 Pseudo-código do pós-processamento (estratégia fix padrão).

```
1: procedure POSTPROCESS(arr, seg, fix)
2:     for i = 0, size(seg)-1 do
3:         for j = seg(i), seg(i+1)-1 do
4:             arr(j) ← arr(j) + fix(i)
```

A Tabela 2.1 ilustra a execução do pré-processamento para um vetor com 12 elementos e 4 segmentos. A primeira linha mostra o vetor original, seguido pelo resultado de cada iteração do algoritmo nas linhas subsequentes. Na primeira iteração, os elementos do segmento 0 foram subtraídos pelo menor elemento do segmento 0. Ainda na primeira iteração o maior e menor elemento do primeiro e segundo segmento, respectivamente, estão identificados. A subtração entre eles é armazenada e utilizada para atualizar o segundo segmento, como mostrado na segunda iteração. Esses passos são repetidos para todos os outros segmentos.

Tabela 2.1: Exemplo do pré-processamento da estratégia *fix* padrão.

it	Segmento 0			Segmento 1			Segmento 2			Segmento 3		
	45	32	76	90	18	39	21	55	62	24	87	16
1st	13	0	**44**	90	**18**	39	21	55	62	24	87	16
2nd	13	0	44	**116**	44	65	**21**	55	62	24	87	16
3rd	13	0	44	116	44	65	116	150	**157**	24	87	**16**
4th	13	0	44	116	44	65	116	150	157	165	228	157
fix	-32			+26			+95			+141		

Os valores armazenados no vetor de *fix* gerado por esse exemplo estão identificados na última linha da tabela e são os valores utilizados para ajustar cada segmento. Depois disso, um algoritmo de ordenação tradicional é chamado, resultando na primeira linha da Tabela 2.2. O pós-processamento então adiciona aos elementos do vetor os valores armazenados no vetor de *fix*, para restaurar seus elementos.

Tabela 2.2: Exemplo do pós-processamento da estratégia *fix* padrão.

it	Segmento 0			Segmento 1			Segmento 2			Segmento 3		
	0	13	44	44	65	116	116	150	157	157	165	228
	↓ +32			↓ -26			↓ -95			↓ -141		
1st	32	45	76	18	39	90	21	55	62	16	24	87

Uma vez que os valores do vetor crescem a cada iteração do pré-processamento, o número de segmentos tratáveis por essa estratégia é limitado. Esse crescimento é principalmente causado pelo que chamamos de **tensão**, que é a diferença entre o maior e menor valor de cada segmento.

Suponha que o maior valor de um certo tipo inteiro seja K, que $V > 0$ seja o maior elemento do vetor e que s seja o número de segmentos deste vetor. No pior caso, todos os segmentos contêm os valores 0 e V, portanto possuem tensão V. Isso significa que, para cada segmento $i \in [0, s-1]$, o pré-processamento adiciona $i \cdot V$ ao i-ésimo elemento. Dessa forma, ao segmento $s-1$, cujo maior elemento é V, será adicionado o valor $((s-1) \cdot V)$. Logo, o maior elemento atualizado do vetor será igual a $s \cdot V$. Como o tipo mencionado é limitado ao valor K, o número de segmentos que podemos ordenar por meio dessa estratégia é pelo menos $S = \lfloor \frac{K}{V} \rfloor$.

A Tabela 2.3 mostra um exemplo dessa limitação para um tipo numérico representado por 8 *bits* ($K = 2^8 - 1 = 255$) em um vetor com o maior elemento igual a $V = 127$. Podemos facilmente encontrar um limite inferior para o número de segmentos calculando $S = \lfloor \frac{255}{127} \rfloor = 2$.

Tabela 2.3: Exemplo da limitação do pré-processamento para um número de 8 *bits*.

it	Segmento 0		Segmento 1		Segmento 2	
	127 01111111	31 00011111	63 00111111	4 00000100	7 00000111	108 01101100
1st	**96** 01100000	0 00000000	63 00111111	**4** 00000100	7 00000111	108 01100101
2nd	96 01100000	0 00000000	**155** 10011011	96 01100000	**7** 00000111	108 01100101
3rd	96 01100000	0 00000000	155 10011011	96 01100000	155 10011011	256 **overflow**

Como esperado, somos capazes de agrupar dois dos três segmentos. Note que o terceiro segmento não poderia ser agrupado porque um *overflow* ocorreria quando o valor 108 fosse ajustado. Notamos que, mesmo que não seja possível tratar todos os segmentos em uma única chamada, a estratégia pode ser usada várias vezes em grupos de segmentos, reduzindo o número de vezes que o algoritmo de ordenação tradicional tem que ser chamado.

Podemos paralelizar parte do primeiro passo do pré-processamento atribuindo uma *thread* a cada segmento ou fazendo uma redução paralela. Para isso, basta encontrar o menor valor de cada segmento e subtraí-lo dos valores dos elementos do segmento. Porém, encontrar o maior valor do segmento da esquerda pode impor uma execução sequencial. Deixamos essa exploração adicional como trabalho futuro e focamos em uma estratégia alternativa, discutida na próxima seção.

Os passos do pós-processamento são completamente independentes. Logo, podem ser executados em paralelo pela simples atribuição de uma *thread* por elemento ou por segmento.

2.2.2 Estratégia Fix Paralela

Embora a estratégia *fix* geral seja capaz de agrupar um número de segmentos maior que o pior caso, ela não é trivialmente paralelizável. Isso nos motivou a olhar uma abordagem diferente, que não permite agrupar tantos segmentos quanto o primeiro, mas é completamente paralelizável. Essa estratégia não considera os valores armazenados no vetor, mas usa o índice do segmento como um prefixo em seus elementos. A vantagem desse método em relação ao geral é que esse não requer o uso de um vetor auxiliar nem a inspeção dos segmentos anteriores para atualizar o segmento corrente.

O Algoritmo 3 apresenta o pseudocódigo do pré-processamento. Primeiro, no caso de não ser conhecido, ele identifica o índice do *bit* mais significativo (MSB - *Most Significant Bit*) do maior valor armazenado no vetor. Então, ele desloca o índice do segmento MSB *bits* para a esquerda e adiciona esse valor em todos os elementos do segmento. Para encontrar o maior elemento do vetor, utilizamos redução paralela e para encontrar o *bit* mais significativo desse elemento, aplicamos deslocamentos para direita. O Algoritmo 4 mostra o pseudocódigo do pós-processamento, que também é trivialmente paralelizável.

Algoritmo 3 Pseudocódigo para o pré-processamento do vetor (estratégia *fix* paralela).

```
1: procedure PREPROCESS(arr, seg)
2:     maxIndex ← index of the most significant bit
3:     for i = 1, size(seg)-1 do
4:         fix ← shiftLeft(i, maxIndex)
5:         for j = seg(i), seg(i+1)-1 do
6:             arr(j) ← arr(j) + fix
```

Suponha que o maior valor de um tipo inteiro possa ser representado por *D bits*. Assuma que *B* é o número de *bits* usado para representar o maior valor do vetor. Como agrupamos os segmentos usando os *bits* mais significativos não utilizados por nenhum dos elementos do vetor, conseguimos agrupar no

Algoritmo 4 Pseudocódigo para o pós-processamento do vetor (estratégia *fix* paralela).

```
1: procedure POSTPROCESS(arr, seg, maxIndex)
2:     for i = 1, size(seg)-1 do
3:         fix ← shiftLeft(i, maxIndex)
4:         for j = seg(i), seg(i+1)-1 do
5:             arr(j) ← arr(j) - fix
```

máximo $S = 2^{D-B}$ segmentos. A Tabela 2.4 mostra um exemplo em que $D = 6$ e $B = 4$. O identificador do segmento, que foi adicionado a cada elemento, está destacado em negrito. Logo, o limitante superior para o número de segmentos é $S = 2^{6-4} = 4$.

Tabela 2.4: Exemplo do preprocessamento *fix sort* paralelo.

it	Segmento 0	Segmento 1	Segmento 2	Segmento 3
	15 8 001111 001000	9 4 001001 000100	7 11 000111 001011	2 14 000010 001110
1st	15 8 **00**1111 **00**1000	9 4 **01**1001 **01**0100	7 11 **10**0111 **10**1011	2 14 **11**0010 **11**1110

A paralelização desse método foi realizada por meio dos pseudocódigos apresentados a seguir. O Algoritmo 5 descreve os passos do pré-processamento. Uma redução paralela é realizada para encontrar o maior elemento do vetor (passo 2), o bit mais significativo (MSB) desse elemento é calculado (passo 3) e, posteriormente, a função *transform* ajusta os elementos do vetor em paralelo (passo 4). Essa função invoca o procedimento PLUS, que calcula os novos valores do vetor. Primeiro o procedimento calcula o valor do *fix* para cada segmento fazendo MSB deslocamentos à esquerda no índice do segmento (passo 6) e depois adiciona esse valor (*fix*) a todos os elementos pertencentes àquele segmento (passo 7).

Algoritmo 5 Pseudocódigo do pré-processamento de um vetor realizado pela estratégia *fix* paralela.

```
1: procedure PREPROCESSING(arr, seg)
2:     max ← max_element(arr)
3:     index ← log2(arr)+1
4:     transform(arr, seg, PLUS < index > )
5: procedure PLUS(value, segId)
6:     fix ← segId << index
7:     value ← value + fix
```

O Algoritmo 6 descreve os passos do pós-processamento, que consiste em retornar o vetor ao seu valor original, subtraindo o valor do *fix* adicionado no pré-processamento.

Algoritmo 6 Pseudocódigo do pós-processamento de um vetor realizado pela estratégia *fix* paralela.

```
1: procedure POSTPROCESSING(arr, seg, index)
2:     transform(arr, seg, MINUS < index > )
3: procedure MINUS(value, segId)
4:     fix ← segId << index
5:     value ← value - fix
```

2.3 Análise dos Algoritmos

Os melhores resultados da ordenação segmentada foram obtidas por meio de estratégias baseadas em dois algoritmos tradicionais de ordenação: merge sort e radix sort. Nesta seção, analisamos cada estratégia usada nesses algoritmos para ordenação segmentada de vetores.

2.3.1 Radix Sort

O *radix sort* é um algoritmo de ordenação por contagem, uma vez que não compara os elementos do vetor para ordená-lo. A complexidade assintótica desse algoritmo é $O(cn)$, sendo n o número de elementos do vetor e c é o número de *bits* do tipo numérico de seus elementos. As implementações baseadas no *radix sort* são: *radix sort* segmentado (**R**), *fix sort* usando as bibliotecas CUB (**FC**) e Thrust (**FT**).

As implementações do *fix sort* (**FC** e **FT**) executam a ordenação segmentada em três passos. O primeiro é o pré-processamento, que consiste em ajustar os valores do vetor para garantir que todos os elementos de um segmento sejam maiores ou iguais a todos os elementos pertencentes a segmentos à sua esquerda. Então, o algoritmo de ordenação *radix sort* é chamado para ordenar o vetor inteiro e termina sua execução executando o pós processamento, que é responsável por restaurar os elementos do vetor a seus valores originais. Assim, a complexidade assintótica do *fix sort* é composta por três partes:

- Pré-processamento: $O(n)$
- Radix sort: $O(cn)$
- Pós-processamento: $O(n)$

Resultando em $O(n+cn+n) = O(cn)$.

O *radix sort* segmentado (**R**) executa a rotina do *radix sort* de forma independente para cada segmento do vetor. Considerando um vetor de tamanho n, número de segmentos m e, consequentemente, tamanho dos segmentos igual a $\frac{n}{m}$, sua complexidade assintótica é igual ao algoritmo tradicional: $O(mc\frac{n}{m}) = O(cn)$.

Ao ordenar vetores com tamanhos diferentes, sua complexidade é:

$$O\left(c\sum_{i=0}^{m} s_i\right) = O\left(c(s_1+s_2+s_3+...+s_m)\right) = O(cn)$$

com s_i sendo o tamanho do i-ésimo segmento, sendo que $s_1+s_2+s_3+...+s_m = n$, então $O(cn)$. Isso significa que, não importa o número de segmentos, ou o tamanho dos segmentos do vetor, estratégias baseadas no radix sort possuem a mesma complexidade computacional.

2.3.2 Merge Sort

O *merge sort* é um algoritmo de divisão e conquista que separa um vetor com n elementos em duas subsequencias com $\frac{n}{2}$ elementos cada. Então ele ordena essas duas subsequencias recursivamente usando merge sort. Finalmente, ele merge as duas subsequencias ordenadas para gerar o vetor ordenado. Esse processo leva a uma complexidade assintótica de $O(n\log(n))$ (Satish et al., 2010).

As estratégias baseadas no *merge sort* são **merge sort segmentado** (**M**) e **hou segmentado** (**H**). Ambas seguem os mesmos passos do algoritmo merge sort tradicional. Portanto, suas complexidades assintóticas são iguais à ordenação de um único vetor. Porém, para ordenar os valores dentro de cada segmento, seus passos de merge somente troca elementos se eles pertencem ao mesmo segmento. Consequentemente, as rotinas executam o mesmo número de comparação do algoritmo tradicional, mas executam menos trocas entre os elementos.

Dessa forma, o número de segmentos e o tamanho dos segmentos não afetam o número de comparações do merge nem a complexidade assintótica do mesmo. Assim, a complexidade do algoritmo permanece a mesma em cenários com segmentos de tamanhos diferentes.

2.4 Resultados Experimentais

Executamos o *benchmark* em sete máquinas diferentes. As configurações de hardware e software estão especificadas na Tabela 2.5 e as configurações das GPUs estão especificadas na Tabela 2.6.

Todos os tamanhos de vetores e número de segmentos no *benchmark* são potências de dois. Os vetores são compostos por números inteiros sem sinal e possuem de 2^{15} até 2^{27} elementos. O limite inferior deste intervalo é para garantir o mínimo grau de quantidade de dados, e o limite superior do intervalo é devido ao tamanho da memória global da maioria das GPUs. Por isso na

Tabela 2.5: Especificações de hardware e configurações de software de cada máquina.

GPU	Modelo do Processador	Memória RAM (GB)	Sistema Operacional	Versão Gcc
GTX 770	2 x Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz	32	Ubuntu 18.04	7.5.0
GTX Titan	1 x Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz	32	Ubuntu 20.04	8.4.0
Tesla K20	2 x Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz	64	Suse 12-SP1	4.8.5
Tesla K40	2 x Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz	64	Suse 12-SP1	4.8.5
GTX 950	1 x Intel(R) Core(TM) i7-5820K CPU @ 3.30GHz	16	Ubuntu 16.04	8.4.0
Quadro 4000	1 x Intel(R) Xeon(R) CPU E5-1620 v3 @ 3.50GHz	32	Ubuntu 16.04	7.4.0
GTX 1070	1 x Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz	32	Ubuntu 20.04	9.3.0

Tabela 2.6: Especificações das GPUs.

Modelo da Placa de Vídeo	Capacidade Computacional	Memória Global (GB)	Número de SMs	Número de Núcleos por SM	Frequência do Núcleo (MHz)	Frequência da Memória (MHz)	Largura de Banda da Memória (GB/s)	Versão CUDA
GTX 770	3.0	2	8	192	1,202	3,600	224	10.1
GTX Titan	3.5	6	15	192	980	3,500	288	10.2
Tesla K20	3.5	4	13	192	706	2,600	208	10.1
Tesla K40	3.5	12	15	192	745	3,004	288	10.1
GTX 950	5.2	2	6	128	1,240	3,305	105.6	10.2
Quadro 4000	5.2	8	13	128	773	3,000	89.6	10.1
GTX 1070	6.1	8	15	128	1,785	4,004	256	10.2

GTX 770 e na GTX 950 o maior vetor foi limitado em 2^{26} elementos.

O *fix sort* introduziu uma restrição no *benchmark*. Essa estratégia agrupa os segmentos usando os *bits* mais significativos não usados pelos elementos do vetor. Então, o número de segmentos é limitado pelo maior elemento e vice-versa. Uma vez que o número máximo de segmentos do *benchmark* foi limitado em 2^{20} e os números inteiros sem sinal possuem 32 *bits*, os valores do vetor foram limitados em 12 *bits*[3].

Para cada combinação de tamanho do vetor e número de segmentos, geramos aleatoriamente dez vetores diferentes. Cada um foi executado dez vezes, calculamos a média da execução e reportamos a média das médias de todos eles como o tempo de execução. O *benchmark*, bem como os códigos deste trabalho estão disponíveis no github[4].

As seguintes implementações foram comparadas neste trabalho: rotina de ordenação segmentada da biblioteca Modern GPU (MGPU) (**M**); rotina de ordenação segmentada da biblioteca CUB (**R**); chamada da rotina de ordenação da biblioteca Thrust, uma vez para cada segmento do vetor (**MT**); implementação do *fix sort* usando a biblioteca CUB (**FC**); implementação do *fix sort* usando a biblioteca Thrust (**FT**); e a rotina de ordenação segmentada implementada por Hou et al. (2017) (**H**).

Além disso, testes foram realizados usando uma implementação sequencial

[3] Ainda é possível usar a estratégia *fix sort* chamando a rotina várias vezes, mas esses testes não foram executados nesse trabalho.

[4] https://github.com/rafaelfschmid/segsort-2020.git

do *fix sort*, que usa o *quick sort* da biblioteca padrão do C++ (STD) (**FS**) e uma implementação sequencial, que consiste em múltiplas chamadas do *quick sort* uma para cada segmento (**MS**) (Schmid e Cáceres, 2019). Porém, como os resultados dessas implementações foram muito abaixo das implementações em GPU, elas não foram discutidas nesse trabalho.

2.4.1 Comportamento com o Aumento no Número de Segmentos

Nesta seção analisamos os resultados empíricos de cada estratégia quando se altera o número de segmentos, mantendo o tamanho do vetor constante. A Figura 2.1 ilustra como cada estratégia se comporta com o aumento no número de segmentos, considerando um vetor com 2^{25} elementos. É importante notar que, uma vez que o tamanho do vetor se mantém constante, o tamanho dos segmentos diminuem com o aumento no número de segmentos.

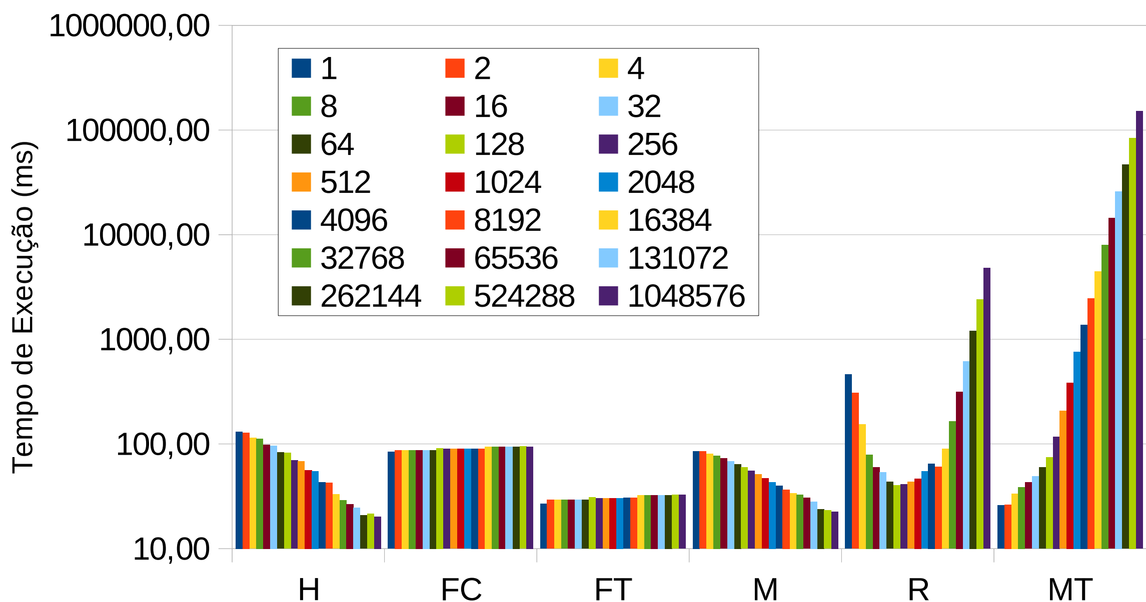


Figura 2.1: Tempo de execução das estratégias com o aumento no número de segmentos em vetores com 2^{25} elementos na Quadro 4000.

Como discutido na Seção 2.3, a complexidade assintótica das estratégias baseadas no *merge sort* não são afetadas pelo número de segmentos, porém, o número de trocas entre os elementos decresce. Portanto, é esperado que o tempo de execução destas estratégias diminuam com o aumento no número de segmentos. A Figura 2.1 mostra este comportamento para ambas as estratégias baseadas no *merge sort*: **M** e **H**, o que é coerente com a análise dos algoritmos.

Dado que as implementações do *fix sort* (**FC** e **FT**) utilizam um algoritmo de ordenação de tradicional, como se houvesse um único segmento, era espe-

rado que o desempenho fosse afetado apenas pelo tamanho do vetor. Apesar disso, os resultados mostraram um pequeno acréscimo no tempo de execução quando o número de segmentos aumenta. Ao analisar o código foi identificado que os passos de pré e pós-processamento do *fix sort* não são afetados quando o número de segmentos muda. Porém, quanto maior o número de segmentos da entrada, mais *bits* devem ser avaliados para colocar os elementos na posição correta. As Tabelas 2.7 e 2.8 mostram dois exemplos que ilustram o porquê do aumento no tempo de execução.

A Tabela 2.7 apresenta um vetor com 8 elementos e 2 segmentos, e a seguinte iteração:

1. Os elementos do vetor são ajustados pelo índice dos segmentos ao qual pertencem.
2. O *bit* menos significativo é usado para colocar os elementos na primeira metade do vetor ou na segunda metade do vetor.
3. O segundo *bit* menos significativo é usado para colocar os elementos na primeira metade do vetor ou na segunda metade do vetor.
4. O terceiro *bit* menos significativo é usado para colocar os elementos na primeira metade do vetor ou na segunda metade do vetor.
5. O quarto *bit* menos significativo é usado para colocar os elementos na primeira metade do vetor ou na segunda metade do vetor.
6. Como os últimos dois *bits* são zero para todos os elementos, não haverá mais mudanças
7. Os elementos do vetor são restaurados para o valor original.
8. Vetores ordenados por segmentos.

A Tabela 2.8 apresenta um vetor com 8 elementos e 4 segmentos, e a seguinte iteração:

1. Os elementos do vetor são ajustados pelo índice dos segmentos ao qual pertencem.
2. O *bit* menos significativo é usado para colocar os elementos na primeira metade do vetor ou na segunda metade do vetor.
3. O segundo *bit* menos significativo é usado para colocar os elementos na primeira metade do vetor ou na segunda metade do vetor.
4. O terceiro *bit* menos significativo é usado para colocar os elementos na primeira metade do vetor ou na segunda metade do vetor.

Tabela 2.7: Exemplo da estratégia *fix sort* ao ordenar um vetor com 8 elementos e 2 segmentos.

it	Segmento 0				Segmento 1			
*	1	7	0	5	4	2	6	3
	000001	000111	000000	000101	000100	000010	000110	000011
1st	000001	000111	000000	000101	001100	001010	001110	001011
2nd	000000	001100	001010	001110	000001	000111	000101	001011
3rd	000000	001100	000001	000101	001010	001110	000111	001011
4th	000000	000001	001010	001011	001100	000101	001110	000111
5th	000000	000001	000101	000111	001010	001011	001100	001110
6th	000000	000001	000101	000111	001010	001011	001100	001110
7th	000000	000001	000101	000111	001010	001011	001100	001110
*	0	1	5	7	2	3	4	6
	000000	000001	000101	000111	000010	000011	000100	000110

5. O quarto *bit* menos significativo é usado para colocar os elementos na primeira metade do vetor ou na segunda metade do vetor.

6. O quinto *bit* menos significativo é usado para colocar os elementos na primeira metade do vetor ou na segunda metade do vetor.

7. Como o último *bit* é zero para todos os elementos, não haverá mais mudanças.

8. Os elementos do vetor são restaurados para o valor original.

9. Vetores ordenados por segmentos.

Tabela 2.8: Exemplo da estratégia *fix sort* ao ordenar um vetor com 8 elementos e 4 segmentos.

it	Segmento 0		Segmento 1		Segmento 2		Segmento 3	
*	1	7	0	5	4	2	6	3
	000001	000111	000000	000101	000100	000010	000110	000011
1st	000001	000111	001000	001101	010100	010010	011110	011011
2nd	001000	010100	010010	011110	000001	000111	001101	011011
3rd	001000	010100	000001	001101	010010	011110	000111	011011
4th	001000	000001	010010	011011	010100	001101	011110	000111
5th	000001	000111	010010	010100	001000	001101	011011	011110
6th	000001	000111	001000	001101	010010	010100	011011	011110
7th	000001	000111	001000	001101	010010	010100	011011	011110
8th	000001	000111	001000	001101	010010	010100	011011	011110
*	1	7	0	5	2	4	3	6
	000001	000111	001000	001101	010010	010100	011011	011110

Esses dois exemplos mostram que duplicando o número de segmentos, o número de iterações do *radix sort* aumenta em 1. Apesar de não ser uma mudança significativa, a GPU demanda por sincronização entre os blocos depois

de cada iteração do radix sort. Dessa forma, mais segmentos significam mais trocas entre blocos e, consequentemente, impacto no tempo de execução.

Apesar da análise assintótica indicar que o número de segmentos não interfere no tempo de execução de **R**, a Figura 2.1 mostra grande impacto em seu tempo de execução. Ao implementar programas em CUDA, ambos, o algoritmo e o mapeamento da computação nos processadores paralelos da GPU são importantes. Examinando o código, foi identificado que essa implementação atribui um segmento para cada bloco CUDA. Por um lado, essa estratégia subestima a capacidade da GPU ao ordenar segmentos pequenos, e por outro, ela superestima a capacidade da GPU ao ordenar segmentos muito grandes. Porém, há um intervalo em que essa implementação se encaixa na dimensão do problema, e executa de forma eficiente.

MT invoca uma rotina de ordenação para cada segmento do vetor, dessa forma, sua complexidade assintótica não é alterada pelo número de segmentos. Porém, quanto maior o número de segmentos, maior é o tempo de execução dessa estratégia. O que acontece é que a cada chamada de função um *overhead* é introduzido no tempo de execução. Esse *overhead* pode se tornar maior que o próprio tempo de realizar a ordenação dos segmentos, fazendo com que essa estratégia se torne extremamente ineficaz.

2.4.2 Análise da Curva-S

Na seção anterior apresentamos o comportamento de cada estratégia ao mudar o número de segmentos. Foi mostrado que uma abordagem pode ser muito eficiente em alguns cenários como também desempenhar muito mal em outros. Esta subseção discute o quão lenta uma estratégia pode ser quando comparada com a estratégia mais rápida em cada cenário.

A curva-S, da Figure 2.2, apresenta o *slowdown* de cada estratégia considerando os cenários em todas as GPUs e todas as dimensões, com exceção das estratégias **MT** e **R**, cujos piores tempos foram omitidos para não poluir o gráfico. Os valores desse gráfico são gerados dividindo o tempo que uma estratégia leva para ordenar um vetor em um cenário específico pelo tempo que a abordagem mais rápida leva para ordenar o vetor no mesmo cenário. Os valores são ordenados independentemente para cada estratégia, gerando a curva-S.

Como os *slowdowns* são ordenados independentemente, não é possível identificar para um determinado ponto no gráfico a dimensão do vetor ou segmento correspondente. Porém, o gráfico da curva-S não é projetado para comparar pontos específicos, seu objetivo é mostrar o quanto o desempenho de uma implementação específica pode divergir da melhor implementação.

A curva-S do **FT** tem o menor *slowdown* no gráfico. Isso significa que

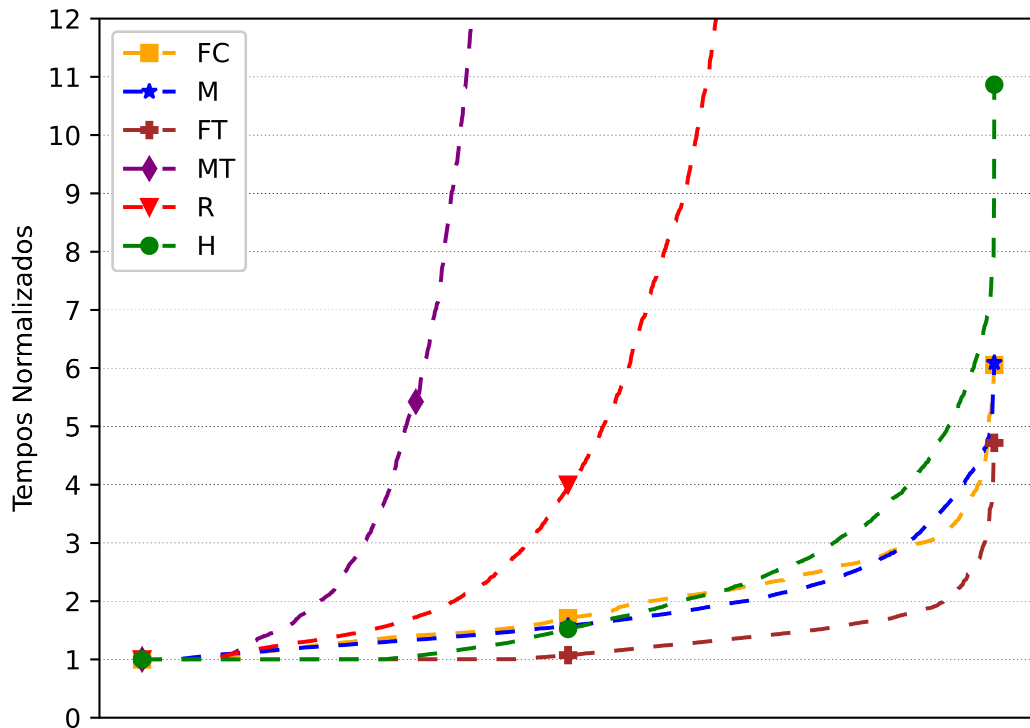


Figura 2.2: Curva-S das estratégias considerando os resultados de todas as GPUs.

quando essa implementação não é a melhor, ela está próxima da melhor na maioria dos casos. No pior caso, considerando todas as GPUs, essa implementação é 4,62 vezes pior que a melhor. Mas ela é menor que 4 em 99,83% dos casos, e melhor que 2 em 94,25% dos casos, como apresentado na Tabela 2.9. Enquanto o pior caso do **FC**, a outra implementação do *fix sort*, é 6,05 pior que o melhor, e menor ou igual a 2 em 59,64% dos casos.

Tabela 2.9: Percentual de casos onde cada estratégia está dentro do *slowdown* máximo especificado quando comparadas com a estratégia mais rápida.

Slowdown Máximo	Percentual de Casos					
	FC [%]	**M** [%]	**FT** [%]	**MT** [%]	**R** [%]	**H** [%]
$= 1.0\times$	8.45	4.21	44.11	8.59	8.19	28.16
$\leq 1.5\times$	41.67	45.50	79.36	15.94	26.74	49.48
$\leq 2.0\times$	59.64	71.40	94.25	21.11	36.67	63.30
$\leq 3.0\times$	91.70	90.04	98.95	25.96	45.18	81.27
$\leq 4.0\times$	98.32	96.28	99.83	29.21	50.12	90.21

A estratégia **R** é a melhor em casos muito específicos (8,19% dos casos) e é muito ruim em outros. Ela é mais que quatro vezes mais lenta que a

melhor estratégia em aproximadamente 50% dos casos. Além disso, o pior caso do **R** é 2469 vezes pior que a implementação mais rápida. Como discutido anteriormente, isso é devido à configuração ruim dos blocos CUDA.

Chamar uma rotina de ordenação para cada segmento (**MT**) é o melhor quando se ordena poucos segmentos grandes (8,59% dos casos), mas pode ser muito ruim em outros casos. Essa abordagem é mais que quatro vezes mais lenta que a estratégia mais rápida em mais de 70% dos casos. Esse comportamento ruim é devido ao *overhead* da chamada da função quando se está ordenando muitos segmentos. O pior caso do **MT** é 5900 vezes pior que a abordagem mais rápida.

A estratégia **H** é a mais rápida em 28,16% dos casos, que é um pouco mais de 4,21% de casos em que **M** é a mais rápida. Porém, na metade final dos *slowdowns*, **M** é mais próxima da estratégia mais rápida que **H**. Enquanto **H** chega a ser quase 11 vezes mais lenta que a implementação mais rápida, **M** é no máximo 6,09. Ambas implementações são comparadas na próxima seção.

2.4.3 Merge Sort Segmentado (**M**) vs. Hou Segmentado (**H**)

Esta seção compara as duas implementações baseadas no *merge sort* discutidas neste trabalho. Apesar das abordagens *merge sort* segmentado (**M**) e *hou* segmentado (**H**) serem estratégias similares, **H** mostra uma pequena vantagem quando aumenta-se o número de segmentos. Ao analisar o código identificamos que a estratégia **M** usa a mesma abordagem independente do tamanho dos segmentos, enquanto a estratégia **H** possui treze implementações diferentes que são chamadas de acordo com o tamanho do segmento, variando de 1 até 2048 elementos:

1. tamanho do segmento ≤ 1
2. $1 <$ tamanho do segmento ≤ 2
3. $2 <$ tamanho do segmento ≤ 4
4. $4 <$ tamanho do segmento ≤ 8
5. $8 <$ tamanho do segmento ≤ 16
6. $16 <$ tamanho do segmento ≤ 32
7. $32 <$ tamanho do segmento ≤ 64
8. $64 <$ tamanho do segmento ≤ 128
9. $128 <$ tamanho do segmento ≤ 256
10. $256 <$ tamanho do segmento ≤ 512

11. 512 < tamanho do segmento ≤ 1024

12. 1024 < tamanho do segmento ≤ 2048

13. tamanho do segmento > 2048

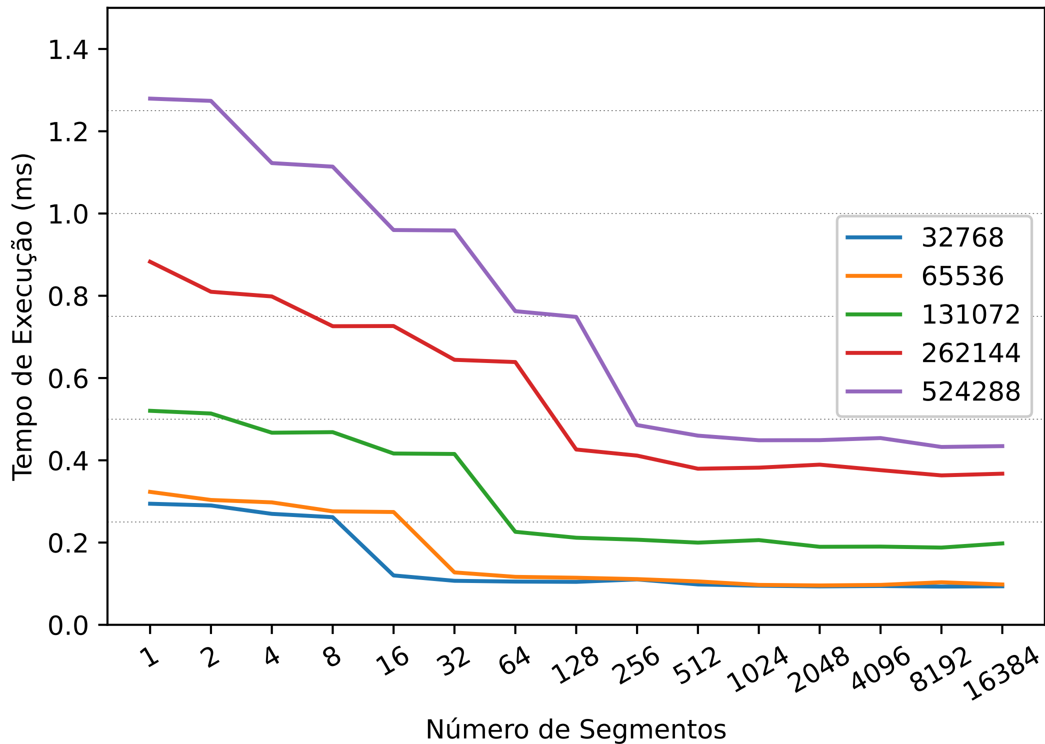


Figura 2.3: Comportamento da estratégia **H** considerando diferentes números de segmentos e tamanhos de vetores na GTX 1070.

Os casos em que a estratégia **H** foi melhor que a estratégia **M** foram principalmente nos casos em que o tamanho dos segmentos são menores ou iguais a 2048. A Figura 2.3 mostra o comportamento desta implementação, na GTX 1070, quando o número de segmentos alcança 2048, para algumas dimensões do vetor: 32768, 65536, 131072, 262144 e 524288. Nos pontos em que o tamanho do segmento atinge 2048, há uma queda abrupta no tempo de execução. Note que o eixo x do gráfico é o número de segmentos do vetor, e cada linha é referente a uma dimensão de vetor diferente, logo, as quedas ocorrerão em posições diferentes para cada linha, quando a divisão do tamanho do vetor pelo número de segmentos atingir 2048. Esse comportamento sugere que chamar um *kernel* especializado para cada dimensão do segmento pode otimizar o tempo de execução. Isso sugere que a melhor implementação pode ser alcançada criando também implementações específicas para as outras dimensões dos segmentos.

2.4.4 Fix Sort: CUB (**FC**) vs. Thrust (**FT**)

As estratégias **FC** e **FT** divergem na biblioteca que utilizam para executar os passos da ordenação segmentada. A estratégia **FT** usa as rotinas da biblioteca Thrust em todos os passos do algoritmo: para encontrar o maior elemento do vetor, ajustar seus valores no pré e pós processamento, e para efetuar a ordenação. Por outro lado, a estratégia **FC** usa a biblioteca CUB para encontrar o maior elemento do vetor e para efetuar a ordenação. Não foi encontrada uma rotina para executar o pré e o pós processamento por essa biblioteca, então foi implementado um *kernel* simples para isso.

A Figura 2.4 apresenta a média, o mínimo e o máximo do *speedup* dos passos do *fix sort* usando a biblioteca Thrust (**FT**) quando comparado com o *fix sort* usando a biblioteca CUB (**FC**) para todas as GPUs e dimensões de vetores. Uma vez que o número de segmentos tem impacto pequeno no tempo de execução, esse gráfico foi simplificado traçando apenas as curvas de vetores com 16384 segmentos, O *speedup* para outros tamanhos de segmentos são muito similares.

Neste gráfico existem nove linhas de *speedup* comparando as estratégias **FT** e **FC**, para um vetor com 16384 segmentos:

1. **Min Fix Sort FC/FT**: *speedup* mínimo da **FT** sobre a **FC** para cada dimensão do vetor.

2. **Min Fix FC/FT**: *speedup* mínimo das etapas de ajuste da **FT** sobre a **FC** para cada dimensão do vetor.

3. **Min Sort FC/FT**: *speedup* mínimo da operação de ordenação da **FT** sobre a **FC** para cada dimensão do vetor.

4. **Max Fix Sort FC/FT**: *speedup* máximo da **FT** sobre a **FC** para cada dimensão do vetor.

5. **Max Fix FC/FT**: *speedup* máximo das etapas de ajuste da **FT** sobre a **FC** para cada dimensão do vetor.

6. **Max Sort FC/FT**: *speedup* máximo da operação de ordenação da **FT** sobre a **FC** para cada dimensão do vetor.

7. **Average Fix Sort FC/FT**: média do *speedup* da **FT** sobre a **FC** para cada dimensão do vetor.

8. **Average Fix FC/FT**: média do *speedup* das etapas de ajuste da **FT** sobre a **FC** para cada dimensão do vetor.

9. **Average Sort FC/FT**: média do *speedup* da operação de ordenação da **FT** sobre a **FC** para cada dimensão do vetor.

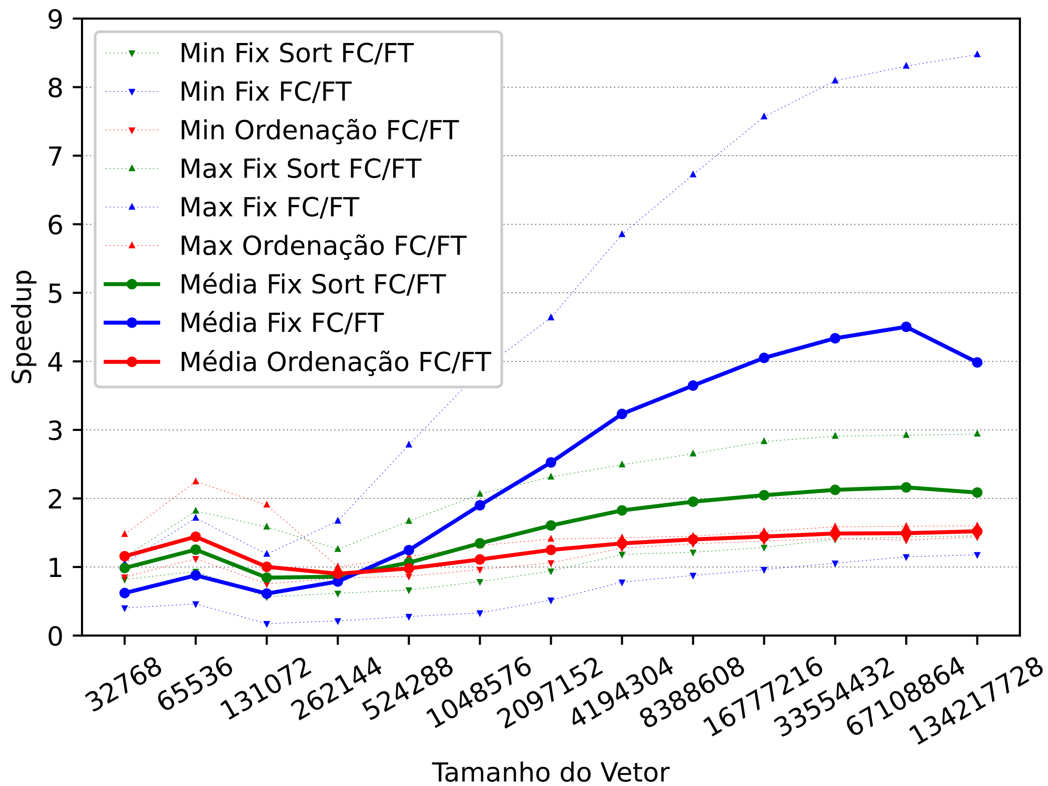


Figura 2.4: *Speedup* da estratégia **FT** quando comparada à estratégia **FC** considerando somente o passo de ajuste (*fix*), somente o passo de ordenação (*sort*), e todo o processo *fix sort*.

A média do *speedup* da estratégia **FT** sobre a estratégia **FC** mostra que a primeira é a melhor opção na maioria dos casos, quando tratamos com grandes segmentos. Os passos de ajuste do vetor são os principais responsáveis por esse resultado, já que aumentam o impacto no tempo de execução da estratégia **FC** com o aumento no tamanho do vetor. Como discutido anteriormente, esses passos são executados por um *kernel* CUDA simples na estratégia **FC**, o que resulta em um *speedup* de até 8,47 vezes da estratégia **FT** sobre a estratégia **FC**, como apresentado pela linha *Max Fix FC/FT*.

Quando o tamanho dos vetores aumenta, **FT** reduz o percentual de tempo gasto nos passos de pré e pós processamento, enquanto a estratégia **FC** aumenta esse percentual. Porém, para vetores grandes, a estratégia **FC** será mais afetada pelos passos de ajuste que a estratégia **FT** e explica porque a primeira é melhor para vetores pequenos e a segunda para vetores grandes.

Embora o impacto do passo de ordenação seja similar para **FT** e **FC**, *Average Sort FC/FT* mostra que a estratégia **FT** é mais rápida que **FC** mesmo para a etapa de ordenação, o que sugere que a biblioteca Thrust é superior à biblioteca CUB para vetores grandes.

O *speedup* mínimo da etapa de ajuste é menor que 1 na maioria dos casos, como mostra a linha *Min Fix Sort FC/FT*. Depois de analisar esses casos, foi identificado que as GPUs K20 e K40 são responsáveis por todas as linhas *Min* do gráfico. Para a maioria das dimensões dos vetores essas GPUs são capazes de realizar os passos de ajuste usando um *kernel* CUDA simples melhor que outros métodos mais sofisticados, como o utilizado na implementação da biblioteca Thrust.

2.4.5 Guia para Escolha da Implementação

Visando prover um guia aos desenvolvedores, apresentamos nesta seção alguns mapas de calor que validam as conclusões obtidas nas seções anteriores. O mapa de calor apresentado nas Tabelas 2.10, 2.11, 2.12, 2.13, 2.14 e 2.15 são indexados pelo tamanho do vetor nas colunas e número de segmentos nas linhas. Os mapas mostram, para cada cenário, o percentual de tempo que cada estratégia foi a melhor considerando os resultados de todas as GPUs.

MT tem bom desempenho somente nos cenários com poucos segmentos. Quando se tem um pouco mais de segmentos, mas ainda, poucos segmentos, a estratégia mais rápida é geralmente a **FT** para vetores grandes e a **FC** para vetores pequenos. A estratégia **H** e, com uma extensão menor **M**, são as estratégias mais rápidas para cenários com grande quantidade de segmentos. **R** é mais rápido quando o tamanho dos segmentos se encaixam na configuração de blocos CUDA da GPU.

Estes resultados mostram que nenhuma estratégia é a melhor em todos os cenários, e a melhor estratégia para um cenário específico pode mudar de uma GPU para outra. Isso encorajou a criação do mapa de calor apresentado na Tabela 2.16, com a estratégia recomendada em cada cenário. A estratégia recomendada nesta tabela foi gerada com o objetivo de minimizar o impacto da escolha errada quando usadas GPUs diferentes. Para isso, foi computado o pior slowdown entre todas as GPUs de cada estratégia para cada cenário e esse valor foi comparado entre as estratégias. Aquela que obteve o melhor resultado foi a estratégia recomendada naquele cenário.

As implementações baseadas no *merge sort* (**H** e **M**) reduzem seus tempos de execução com o aumento no número de segmentos, enquanto as implementações do *fix sort* aumentam sutilmente seus tempos de execução. Isso justifica porque as estratégias baseadas no *merge sort* aparecem somente na parte triangular inferior esquerda da tabela, e as estratégias do *fix sort* aparecem principalmente na parte triangular superior direita da tabela.

Considerando a Tabela 2.16 para escolher a implementação para cada cenário, a estratégia **FT** é a escolha recomendada em 50,40% dos casos, seguido

Tabela 2.10: Percentual de vezes em que a estratégia **FT** é a mais rápida em cada combinação de tamanho do vetor e número de segmentos, considerando todas as GPUs.

	Tamanho do Vetor (2^n)												
Número de Segmentos (2^m)	15	16	17	18	19	20	21	22	23	24	25	26	27
* 0	–	–	–	–	–	–	–	43%	7%	14%	21%	14%	21%
1	–	64%	14%	–	14%	29%	43%	57%	57%	57%	43%	43%	43%
2	7%	86%	14%	14%	50%	71%	86%	93%	64%	71%	57%	50%	43%
3	–	57%	14%	7%	57%	71%	86%	100%	93%	79%	57%	57%	57%
4	–	43%	7%	7%	43%	64%	86%	100%	100%	100%	93%	64%	57%
5	7%	29%	7%	–	36%	57%	79%	100%	100%	100%	100%	93%	57%
6	7%	29%	14%	–	29%	57%	79%	100%	100%	100%	100%	100%	71%
7	–	29%	14%	7%	29%	50%	71%	93%	100%	100%	100%	100%	71%
8	–	29%	14%	14%	14%	36%	71%	79%	93%	93%	100%	100%	71%
9	–	36%	14%	14%	29%	21%	57%	79%	86%	93%	100%	100%	71%
10	–	29%	14%	–	7%	36%	64%	93%	86%	93%	100%	100%	71%
11	–	50%	14%	–	7%	21%	57%	86%	100%	100%	100%	100%	71%
12	–	29%	14%	–	–	7%	21%	71%	79%	100%	100%	100%	71%
13	–	36%	14%	–	–	–	21%	57%	71%	86%	100%	100%	71%
14	–	29%	14%	–	–	–	–	29%	43%	64%	79%	93%	71%
15	–	14%	14%	–	–	–	–	14%	21%	43%	64%	71%	64%
16	–	–	14%	–	–	–	–	–	21%	29%	43%	43%	57%
17	–	–	–	–	–	7%	7%	14%	14%	21%	29%	36%	36%
18	–	–	–	–	7%	14%	21%	29%	21%	29%	21%	29%	36%
19	–	–	–	–	–	21%	14%	50%	21%	21%	21%	21%	29%
20	–	–	–	–	–	–	21%	43%	36%	14%	14%	21%	21%

Tabela 2.11: Percentual de vezes em que a estratégia **FC** é a mais rápida em cada combinação de tamanho do vetor e número de segmentos, considerando todas as GPUs.

	Tamanho do Vetor (2^n)												
Número de Segmentos (2^m)	15	16	17	18	19	20	21	22	23	24	25	26	27
* 0	–	–	29%	7%	–	7%	7%	–	–	–	–	–	–
1	43%	14%	43%	64%	43%	29%	14%	–	–	–	–	–	–
2	29%	14%	71%	71%	43%	29%	14%	–	–	–	–	–	–
3	14%	7%	57%	71%	43%	29%	14%	–	–	–	–	–	–
4	7%	–	36%	43%	43%	29%	14%	–	–	–	–	–	–
5	–	–	21%	29%	21%	14%	14%	–	–	–	–	–	–
6	14%	7%	–	29%	21%	14%	7%	–	–	–	–	–	–
7	–	7%	7%	–	14%	14%	7%	–	–	–	–	–	–
8	–	–	7%	7%	–	14%	7%	–	–	–	–	–	–
9	–	–	36%	36%	7%	7%	7%	–	–	–	–	–	–
10	–	–	36%	36%	29%	14%	7%	–	–	–	–	–	–
11	–	–	21%	7%	21%	21%	14%	–	–	–	–	–	–
12	–	–	29%	14%	7%	14%	14%	–	–	–	–	–	–
13	–	–	29%	21%	7%	–	7%	–	–	–	–	–	–
14	–	–	21%	14%	14%	–	–	–	–	–	–	–	–
15	–	–	7%	21%	7%	–	–	–	–	–	–	–	–
16	–	–	7%	21%	7%	–	–	–	–	–	–	–	–
17	–	–	–	36%	21%	21%	7%	–	–	–	–	–	–
18	–	–	–	–	36%	29%	14%	–	–	–	–	–	–
19	–	–	–	–	–	29%	14%	–	–	–	–	–	–
20	–	–	–	–	–	–	7%	–	–	–	–	–	–

Tabela 2.12: Percentual de vezes em que a estratégia **H** é a mais rápida em cada combinação de tamanho do vetor e número de segmentos, considerando todas as GPUs.

	Tamanho do Vetor (2^n)												
Número de Segmentos (2^m)	15	16	17	18	19	20	21	22	23	24	25	26	27
* 0	–	–	–	–	–	–	–	–	–	–	–	–	–
1	–	–	–	–	–	–	–	–	–	–	–	–	–
2	–	–	–	–	–	–	–	–	–	–	–	–	–
3	–	–	–	–	–	–	–	–	–	–	–	–	–
4	21%	–	–	–	–	–	–	–	–	–	–	–	–
5	21%	7%	–	–	–	–	–	–	–	–	–	–	–
6	29%	7%	–	–	–	–	–	–	–	–	–	–	–
7	93%	7%	14%	21%	–	–	–	–	–	–	–	–	–
8	100%	43%	7%	21%	29%	–	–	–	–	–	–	–	–
9	100%	64%	36%	43%	29%	29%	–	–	–	–	–	–	–
10	100%	71%	29%	64%	64%	43%	21%	–	–	–	–	–	–
11	100%	50%	50%	93%	71%	57%	29%	14%	–	–	–	–	–
12	100%	71%	50%	86%	93%	79%	64%	29%	21%	–	–	–	–
13	100%	64%	50%	79%	93%	100%	71%	43%	29%	14%	–	–	–
14	100%	71%	36%	86%	86%	100%	100%	71%	57%	29%	21%	–	–
15	–	86%	57%	79%	93%	100%	100%	86%	79%	50%	36%	29%	–
16	–	–	36%	71%	86%	93%	100%	100%	71%	64%	57%	43%	14%
17	–	–	–	43%	57%	57%	86%	79%	71%	64%	71%	64%	21%
18	–	–	–	–	43%	36%	29%	57%	71%	57%	71%	71%	36%
19	–	–	–	–	–	14%	14%	14%	64%	57%	71%	71%	43%
20	–	–	–	–	–	–	7%	7%	14%	79%	71%	71%	43%

Tabela 2.13: Percentual de vezes em que a estratégia **M** é a mais rápida em cada combinação de tamanho do vetor e número de segmentos, considerando todas as GPUs.

	Tamanho do Vetor (2^n)												
Número de Segmentos (2^m)	15	16	17	18	19	20	21	22	23	24	25	26	27
* 0	–	–	–	–	–	–	–	–	–	–	–	–	–
1	43%	–	–	–	–	–	–	–	–	–	–	–	–
2	21%	–	–	–	–	–	–	–	–	–	–	–	–
3	21%	–	–	–	–	–	–	–	–	–	–	–	–
4	–	–	–	–	–	–	–	–	–	–	–	–	–
5	–	–	–	–	–	–	–	–	–	–	–	–	–
6	–	–	7%	–	–	–	–	–	–	–	–	–	–
7	–	–	7%	–	–	–	–	–	–	–	–	–	–
8	–	–	14%	–	–	–	–	–	–	–	–	–	–
9	–	–	7%	–	–	7%	7%	–	–	–	–	–	–
10	–	–	21%	–	–	–	–	–	–	–	–	–	–
11	–	–	14%	–	–	–	–	–	–	–	–	–	–
12	–	–	7%	–	–	–	–	–	–	–	–	–	–
13	–	–	7%	–	–	–	–	–	–	–	–	–	–
14	–	–	29%	–	–	–	–	–	–	7%	–	7%	–
15	–	–	21%	–	–	–	–	–	–	7%	–	–	7%
16	–	–	43%	7%	7%	7%	–	–	7%	7%	–	14%	–
17	–	–	–	21%	21%	14%	–	7%	14%	14%	–	–	14%
18	–	–	–	–	14%	21%	36%	14%	7%	14%	7%	–	–
19	–	–	–	–	–	36%	57%	36%	14%	21%	7%	7%	–
20	–	–	–	–	–	–	64%	50%	50%	7%	14%	7%	7%

Tabela 2.14: Percentual de vezes em que a estratégia **R** é a mais rápida em cada combinação de tamanho do vetor e número de segmentos, considerando todas as GPUs.

	Tamanho do Vetor (2^n)												
Número de Segmentos (2^m)	15	16	17	18	19	20	21	22	23	24	25	26	27
* 0	–	–	–	–	–	–	–	–	–	–	–	–	–
1	–	–	–	–	–	–	–	–	–	–	–	–	–
2	43%	–	–	–	–	–	–	–	–	–	–	–	–
3	64%	36%	29%	21%	–	–	–	–	–	–	–	–	–
4	71%	57%	57%	50%	14%	7%	–	–	–	–	–	–	–
5	71%	64%	71%	71%	43%	29%	7%	–	–	–	–	–	–
6	50%	57%	79%	71%	50%	29%	14%	–	–	–	–	–	–
7	7%	57%	57%	71%	57%	36%	21%	7%	–	–	–	–	–
8	–	29%	57%	57%	57%	50%	21%	21%	7%	7%	–	–	–
9	–	–	7%	7%	36%	36%	29%	21%	14%	7%	–	–	–
10	–	–	–	–	–	7%	7%	7%	14%	7%	–	–	–
11	–	–	–	–	–	–	–	–	–	–	–	–	–
12	–	–	–	–	–	–	–	–	–	–	–	–	–
13	–	–	–	–	–	–	–	–	–	–	–	–	–
14	–	–	–	–	–	–	–	–	–	–	–	–	–
15	–	–	–	–	–	–	–	–	–	–	–	–	–
16	–	–	–	–	–	–	–	–	–	–	–	–	–
17	–	–	–	–	–	–	–	–	–	–	–	–	–
18	–	–	–	–	–	–	–	–	–	–	–	–	–
19	–	–	–	–	–	–	–	–	–	–	–	–	–
20	–	–	–	–	–	–	–	–	–	–	–	–	–

Tabela 2.15: Percentual de vezes em que a estratégia **MT** é a mais rápida em cada combinação de tamanho do vetor e número de segmentos, considerando todas as GPUs.

	Tamanho do Vetor (2^n)												
Número de Segmentos (2^m)	15	16	17	18	19	20	21	22	23	24	25	26	27
* 0	100%	100%	71%	93%	100%	93%	93%	57%	93%	86%	79%	86%	50%
1	14%	21%	43%	36%	43%	43%	43%	43%	43%	43%	57%	57%	29%
2	–	–	14%	14%	7%	–	–	7%	36%	29%	43%	50%	29%
3	–	–	–	–	–	–	–	–	7%	21%	43%	43%	14%
4	–	–	–	–	–	–	–	–	–	–	7%	36%	14%
5	–	–	–	–	–	–	–	–	–	–	–	7%	14%
6	–	–	–	–	–	–	–	–	–	–	–	–	–
7	–	–	–	–	–	–	–	–	–	–	–	–	–
8	–	–	–	–	–	–	–	–	–	–	–	–	–
9	–	–	–	–	–	–	–	–	–	–	–	–	–
10	–	–	–	–	–	–	–	–	–	–	–	–	–
11	–	–	–	–	–	–	–	–	–	–	–	–	–
12	–	–	–	–	–	–	–	–	–	–	–	–	–
13	–	–	–	–	–	–	–	–	–	–	–	–	–
14	–	–	–	–	–	–	–	–	–	–	–	–	–
15	–	–	–	–	–	–	–	–	–	–	–	–	–
16	–	–	–	–	–	–	–	–	–	–	–	–	–
17	–	–	–	–	–	–	–	–	–	–	–	–	–
18	–	–	–	–	–	–	–	–	–	–	–	–	–
19	–	–	–	–	–	–	–	–	–	–	–	–	–
20	–	–	–	–	–	–	–	–	–	–	–	–	–

Tabela 2.16: Cada cenário com a estratégia que resulta na escolha que gera o menor impacto considerando os slowdowns das estratégias em todas as GPUs.

Número de Segmentos (2^m)	Tamanho do Vetor (2^n) 15	16	17	18	19	20	21	22	23	24	25	26	27
* 0	MT	MT	MT	MT	MT	MT	MT	MT	MT	MT	MT	MT	MT
1	FT	FT	MT	MT	FT	FT	FT	FT	FT	FT	FT	FT	FT
2	FT	FT	FC	FC	FT	FT	FT	FT	FT	FT	FT	FT	FT
3	R	FT	FC	FC	FT	FT	FT	FT	FT	FT	FT	FT	FT
4	R	R	FC	FC	FT	FT	FT	FT	FT	FT	FT	FT	FT
5	R	R	R	R	FC	FT	FT	FT	FT	FT	FT	FT	FT
6	H	R	R	R	R	FT	FT	FT	FT	FT	FT	FT	FT
7	H	FT	R	R	R	R	FT	FT	FT	FT	FT	FT	FT
8	H	H	FC	FC	H	R	FT	FT	FT	FT	FT	FT	FT
9	H	H	FC	H	H	H	FT	FT	FT	FT	FT	FT	FT
10	H	H	FC	H	H	H	FT	FT	FT	FT	FT	FT	FT
11	H	H	H	H	M	H	H	FT	FT	FT	FT	FT	FT
12	H	H	H	H	H	H	H	H	FT	FT	FT	FT	FT
13	H	H	H	H	H	H	H	H	FT	FT	FT	FT	FT
14	H	H	H	H	H	H	H	H	H	FT	FT	FT	FT
15	-	H	H	H	H	H	H	H	H	H	FT	FT	FT
16	-	-	H	H	M	H	H	H	H	H	H	FT	FT
17	-	-	-	M	M	H	H	H	H	H	H	H	FT
18	-	-	-	-	M	M	M	H	H	H	H	H	H
19	-	-	-	-	-	M	M	H	H	H	H	H	H
20	-	-	-	-	-	-	M	M	M	H	H	H	H

pela estratégia **H** que é indicada em 34,92% , a estratégia **R** com 6,35%, a estratégia **M** correspondendo a 5,95% e a estratégia **FC** representando 2,38% do *benchmark*.

As Figuras 2.5, 2.6, 2.7, 2.8, 2.9 e 2.10 mostram o pior impacto ao usar a Tabela 2.16 como guia para executar a ordenação segmentada em cada cenário. Nessas figuras apresentamos os tempos normalizados, anteriormente descritos na Figura 2.2, mas agora considerando somente os cenários nos quais cada estratégia é recomendada em cada gráfico. Por exemplo, a Figura 2.10 foi traçada considerando somente os cenários nos quais a estratégia **H** é apresentada na Tabela 2.16. Note que, para cada figura, a curva-S para a abordagem recomendada é a menor ou muito próxima de 1 na maioria dos casos.

A Tabela 2.17 mostra o impacto das escolhas, ao seguir a estratégia recomendada na Tabela 2.16.

Tabela 2.17: Percentual de casos em que cada estratégia está dentro do *slowdown* máximo especificado, ao seguir as recomendações da Tabela 2.16.

Slowdown Máximo	Percentual de casos **FC** [%]	**M** [%]	**FT** [%]	**MT** [%]	**R** [%]	**H** [%]
$= 1.0\times$	41,56	32,74	79,53	82,52	63,39	70.72
$\leq 1.5\times$	81,82	99,40	99,07	97,57	91,07	98.74
$\leq 2.0\times$	100,00	100,00	99,80	100,00	100,00	99.75
$\leq 3.0\times$	100,00	100,00	100,00	100,00	100,00	100.00

De acordo com a Tabela 2.17, para cenários em que o **FT** é a estratégia recomendada, escolher essa estratégia garante que a escolha será a mais rápida em 79,53% dos casos (primeira linha), e será menos de 1,5 vezes pior que o tempo mais rápido em 99,07% dos casos (segunda linha). É importante destacar que essa recomendação raramente levará a um *slowdown* maior que duas

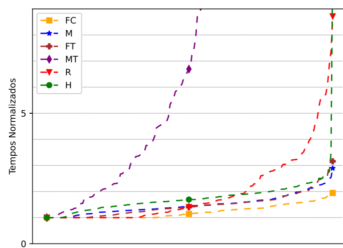


Figura 2.5: Curva-S dos cenários nos quais **FC** é a estratégia com o menor impacto entre todas as GPUs.

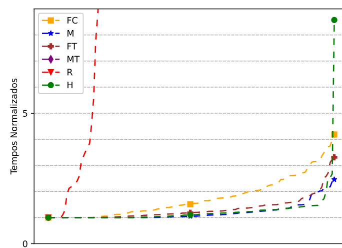


Figura 2.6: Curva-S dos cenários nos quais **M** é a estratégia com o menor impacto entre todas as GPUs.

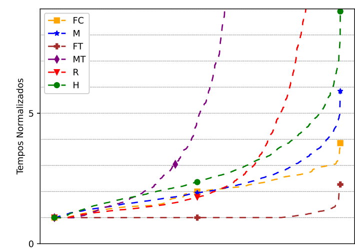


Figura 2.7: Curva-S dos cenários nos quais **FT** é a estratégia com o menor impacto entre todas as GPUs.

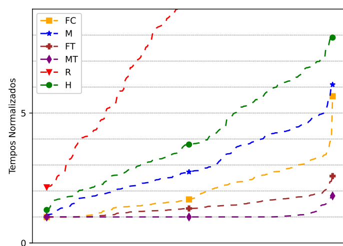


Figura 2.8: Curva-S dos cenários nos quais **MT** é a estratégia com o menor impacto entre todas as GPUs.

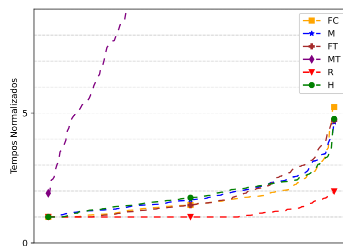


Figura 2.9: Curva-S dos cenários nos quais **R** é a estratégia com o menor impacto entre todas as GPUs.

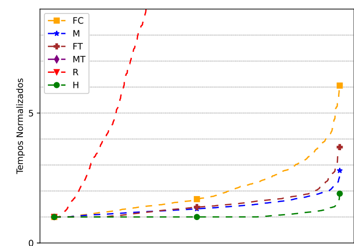


Figura 2.10: Curva-S dos cenários nos quais **H** é a estratégia com o menor impacto entre todas as GPUs.

vezes (terceira linha) e nunca a um *slowdown* maior que três vezes (quarta linha).

2.4.6 Tamanhos de Segmentos: Iguais vs. Diferentes

Todos os testes foram analisados juntos na seção anterior, sem diferenciar experimentos com segmentos de tamanhos iguais e segmentos de tamanhos diferentes. Nesta subseção são analisados os resultados considerando as diferenças entre esses cenários.

A Tabela 2.18 mostra resultados estatísticos de cada estratégia para os testes em que os todos os segmentos do vetor possuem o mesmo tamanho. Já a Tabela 2.19 mostra os resultados de cada estratégia considerando segmentos de tamanhos diferentes. Ambas as tabelas são divididas em duas partes: o percentual de casos em que cada estratégia se encaixa no máximo *slowdown* quando comparado à estratégia mais rápida, e a mediana, média e o maior *slowdown* alcançado caso a estratégia fosse escolhida em todos os cenários.

Tabela 2.18: Percentual de casos onde cada estratégia está dentro do *slowdown* máximo especificado, quando comparada com a estratégia mais rápida considerando somente segmentos do mesmo tamanho.

Slowdown	Percentual de casos					
Máximo	**FC** [%]	**M** [%]	**FT** [%]	**MT** [%]	**R** [%]	**H** [%]
$= 1.0 \times$	5.63	4.88	37.69	8.83	11.38	33.68
$\leq 1.5 \times$	35.89	43.61	71.43	15.97	31.77	53.08
$\leq 2.0 \times$	56.10	69.51	91.06	21.20	41.46	64.87
$\leq 3.0 \times$	88.39	89.90	97.91	25.96	49.30	82.46
$\leq 4.0 \times$	96.81	96.28	99.65	29.15	53.19	91.35
Metric	**FC** [$\times$]	**M** [$\times$]	**FT** [$\times$]	**MT** [$\times$]	**R** [$\times$]	**H** [$\times$]
Median	1.79	1.60	1.17	5.50	3.16	1.40
Average	2.02	1.86	1.37	85.61	49.79	1.96
Max	6.05	6.09	4.62	5927.63	2469.51	7.90

Essas tabelas ilustram que as implementações do fix sort (**FT** and **FC**) são beneficiadas por segmentos de tamanhos diferentes. Enquanto **FT** é a estratégia mais rápida em somente 37% dos casos quando ordena segmentos de tamanhos iguais (Tabela 2.18), ela é a mais rápida em 50,52% dos casos quando ordena segmentos de tamanhos diferentes (Tabela 2.19). Além disso, a média e o máximo *slowdowns* no primeiro caso são 1,37 e 4,62, respectivamente, enquanto que no segundo caso são 1,18 e 2,86. Destaque-se ainda que esses valores são muito menores que a média e o máximo *slowdowns* oferecidos pelas outras abordagens. Como discutido anteriormente, a complexidade assintótica dos algoritmos não são afetadas pelo tamanho dos segmentos, por-

Tabela 2.19: Percentual de casos onde cada estratégia está dentro do *slowdown* máximo especificado, quando comparada com a estratégia mais rápida considerando somente segmentos do mesmo tamanho.

Slowdown Máximo	Percentual de casos					
	FC [%]	**M** [%]	**FT** [%]	**MT** [%]	**R** [%]	**H** [%]
$= 1.0 \times$	11.27	3.54	50.52	8.36	4.99	22.65
$\leq 1.5 \times$	47.44	47.39	87.28	15.91	21.72	45.88
$\leq 2.0 \times$	63.18	73.29	97.44	21.02	31.88	61.73
$\leq 3.0 \times$	95.01	90.19	100.00	25.96	41.06	80.08
$\leq 4.0 \times$	99.83	96.28	100.00	29.27	47.04	89.08
Metric	**FC** [$\times$]	**M** [$\times$]	**FT** [$\times$]	**MT** [$\times$]	**R** [$\times$]	**H** [$\times$]
Median	1.56	1.54	1.00	5.36	4.61	1.63
Average	1.79	1.82	1.19	71.89	43.72	2.13
Max	4.23	6.09	2.86	5056.63	2084.33	7.90

tanto esse comportamento é uma particularidade entre hardware e software, isto é, entre a arquitetura da GPU e as implementações das estratégias.

Como demonstrado no trabalho de Schmid et al. (2016), a rotina do *radix sort* da biblioteca Thrust não é afetada pela disposição dos elementos do vetor. Portanto, as implementações que usam essa rotina (**FT** e **MT**) também não devem ser. Por outro lado, nas implementações baseadas no *merge sort*, quando os segmentos possuem o mesmo tamanho, a tendência é que mais *threads* sejam alocadas em um mesmo bloco CUDA. Ao alterar os testes para segmentos de tamanhos diferentes, observa-se muito mais divergências entre as threads dentro do mesmo bloco CUDA. Assim, as implementações baseadas neste algoritmo devem ser mais afetadas.

2.4.7 Impacto da Escolha de **FT** em cada Modelo de GPU

Esta subseção analisa o impacto da escolha da estratégia **FT** para cada modelo de GPU. A Tabela 2.20 mostra o percentual de casos em que **FT** está limitado pelo *slowdown* especificado, quando comparado com a estratégia mais rápida (colunas 3-6), e sua mediana, média e máximo *slowdowns* (colunas 7-9) em cada modelo de GPU (linhas 1-7 e 9-15). A tabela mostra os resultados para segmentos de tamanhos diferentes (linhas 1-8) e iguais (linhas 9-16). Também é mostrado o resultado geral considerando segmentos de tamanhos iguais e diferentes para todos os modelos de GPU (linhas 8 e 16). Quando ordenando segmentos de diferentes tamanhos na GTX 770 (linha 1), **FT** é a mais rápida em 41,99% dos casos (coluna 3) e é menos que duas vezes mais lenta que a abordagem mais rápida em 99,57% dos casos (coluna 5). Além disso, a média do *slowdown* (coluna 8) e o *slowdown* máximo (coluna 9) são,

Tabela 2.20: Dados estatísticos com o impacto de sempre escolher o *fix sort* para realizar a ordenação segmentada.

	Modelo de GPU	Tamanho dos segmentos	**FT** *slowdown* máximo				**FT** *slowdown*		
			$= 1.0 \times$ [%]	$\leq 1.5 \times$ [%]	$\leq 2.0 \times$ [%]	$\leq 3.0 \times$ [%]	Mediana [$\times$]	Média [$\times$]	Máximo [$\times$]
1	GTX 770	Diferentes	41.99	79.65	99.57	100.00	1.13	1.25	2.01
2	GTX Titan	Diferentes	60.32	81.75	94.84	100.00	1.00	1.22	2.70
3	K20	Diferentes	46.43	86.51	95.24	100.00	1.03	1.21	2.86
4	K40	Diferentes	49.60	79.37	94.44	100.00	1.01	1.23	2.80
5	GTX 950	Diferentes	48.05	95.67	99.57	100.00	1.02	1.14	2.01
6	Quadro 4000	Diferentes	64.68	94.44	99.60	100.00	1.00	1.10	2.24
7	GTX 1070	Diferentes	41.67	93.65	99.21	100.00	1.09	1.17	2.09
8	Todos os Modelos	Diferentes	50.52	87.28	97.44	100.00	1.00	1.19	2.86
9	GTX 770	Iguais	29.87	52.38	77.92	99.13	1.39	1.55	3.15
10	GTX Titan	Iguais	45.63	71.43	89.29	96.83	1.04	1.38	4.44
11	K20	Iguais	31.35	78.17	89.68	94.84	1.27	1.42	4.17
12	K40	Iguais	42.46	75.40	87.70	94.84	1.07	1.39	4.62
13	GTX 950	Iguais	35.50	70.13	98.27	100.00	1.19	1.30	2.22
14	Quadro 4000	Iguais	43.65	84.13	95.63	100.00	1.04	1.23	2.55
15	GTX 1070	Iguais	34.52	66.67	98.41	100.00	1.32	1.34	2.57
16	Todos os modelos	Iguais	37.69	71.43	91.96	97.91	1.17	1.37	4.62

respectivamente, 1,25 e 2,01.

Apesar de os resultados diferirem em cada modelo de GPU, o resultado individual (linhas 1-7) são consistentes com os resultados gerais (linha 8) ao lidar com segmentos de tamanhos diferentes. Neste caso, *a)* a média dos *slowdowns* em todas as GPUs (1,10 até 1,25) estão a no máximo 1,77 desvios padrões da média (1,19); *b)* os maiores *slowdowns* (2,01 até 2,86) estão a no máximo 1,33 desvios padrões da média (2,39); *c)* a estratégia **FT** é no máximo 1,5 vezes pior que a abordagem mais rápida em pelo menos 79,37% dos casos em cada uma das GPUs; e *d)* a estratégia **FT** é a mais rápida em 50,52% dos casos quando considerados todos os modelos de GPUs.

Ao comparar segmentos de mesmo tamanho (linhas 9-15), é observado que a média dos *slowdowns* na GTX 770 (linha 9) é maior que nas outras, resultado que deve ser investigado em trabalho futuro. No entanto, os resultados são consistentes com os resultados gerais (linha 16). Neste caso, *a)* a média dos *slowdowns* em todas as GPUs (1,23 até 1,55) estão a no máximo 1,90 desvios padrões da média (1,37); *b)* os maiores *slowdowns* (2,22 até 4,62) estão a no máximo 1,33 desvios padrões da média. (3,39); *c)* a estratégia **FT** é no máximo 1,5 vezes pior que a abordagem mais rápida em pelo menos 52,39% dos casos em cada uma das GPUs; e *d)* a estratégia **FT** é a mais rápida em 37,69% dos casos quando considerados todos os modelos de GPUs.

2.5 Conclusões e Trabalhos Futuros

Muitos problemas reais requerem uma implementação eficiente de ordenação segmentada, uma rotina que consiste em ordenar vários segmentos individuais de vetores. Neste capítulo, foram avaliadas as implementações mais rápidas conhecidas baseadas em GPU, em vários modelos de GPUs diferentes.

Foram analisados seus desempenhos quando ordenam diferentes números de segmentos e tamanhos de segmentos e na ordenação de segmentos de tamanhos iguais e diferentes.

Primeiro, foi avaliado como o número de segmentos afeta o desempenho de cada implementação e discutido como a complexidade assintótica e os detalhes de implementação causam o desempenho observado.

Depois, foi realizada uma análise da curva-S e mostrado que, mesmo que cada estratégia possa ser rápida em algumas dimensões do problema (número de segmentos, tamanho dos segmentos, ou segmentos de tamanhos iguais e diferentes) algumas abordagens podem causar *slowdowns* muito altos (até 2469 vezes mais lento) em configurações específicas. Esta análise também revelou que, em geral, a estratégia **FT** é a opção que impacta menos no tempo de execução, além de ser a abordagem mais veloz em 44,11% dos cenários. Além disso, o maior *slowdown* causado pela estratégia **FT** foi 4,62 vezes, muito menor que o máximo *slowdown* observado em outras abordagens.

Em seguida, foram analisados os mapas indicando que para cada configuração do problema a estratégia que desempenhava bem e foi proposto um mapa que recomenda a estratégia mais rápida para várias dimensões de entrada. Também foi realizada uma análise na curva-S para avaliar a média e o maior impacto no desempenho ao selecionar a abordagem sugerida por esse mapa. Essa análise mostrou que, seguindo a estratégia recomendada, o maior *slowdown* é menos de três vezes mais lento que a abordagem mais rápida e menos de 1,5 vezes pior na maioria dos casos.

Depois disso, cada estratégia foi avaliada considerando cenários com segmentos de tamanhos iguais e diferentes, e os resultados mostraram que a estratégia **FT** supera as outras abordagens principalmente quando ordena segmentos de tamanhos diferentes. Neste caso, ela é mais rápida que as outras estratégias em 50,52% dos cenários e é menos de duas vezes mais rápida que a abordagem mais rápida em 97,44% dos casos.

Finalmente, é avaliado o impacto de se escolher a abordagem **FT** para cada GPU, que mostrou que, mesmo que os resultados difiram em cada modelo de GPU, os resultados individuais (por modelo de GPU) são consistentes com os resultados gerais (incluindo todos os modelos de GPU).

Os experimentos realizados neste trabalho consideram somente vetores inteiros com valores não-negativos. A estratégia geral do *fix sort* naturalmente transforma números negativos em não negativos quando aplica o fator *fix*. Note que, uma vez que a variável *prevMax* é inicializada com zero (linha 2 do Algoritmo 1), o fator *fix* do primeiro segmento será $-curMin$ (o menor elemento do segmento i) e, consequentemente, o menor elemento do primeiro segmento será ajustado para zero.

Como trabalho futuro pretendemos projetar uma nova implementação do *fix sort* que consista da adição de cada elemento do vetor pelo maior elemento multiplicado pelo índice do segmento. Dessa forma, a implementação compreenderá também outros tipos numéricos, como double e float. Schmid e Cáceres (2019) implementaram o *fix sort* paralelo, usando o *radix sort* da biblioteca Thrust e alcançaram um dos melhores resultados entre todas as implementações. O pré e o pós-processamento são responsáveis por pelo menos 18% do tempo de execução do *fix sort*, isto é, mesmo se o *fix sort* tiver que ajustar o vetor, executar a ordenação e depois reajustá-lo novamente, ele ainda foi uma das melhores estratégias. Esse resultado sugere que para obtermos implementações mais eficientes, o *fix sort* pode ser utilizado nessa tentativa. Além disso, outras dimensões para o problema devem ser testadas, como por exemplo, vetores com tamanho menores que 32768.

Dentre as várias aplicações de ordenação segmentada de vetores, uma de particular interesse para o nosso trabalho foi a sua utilização na heurística min-min. Essa heurística é utilizada para resolver, de forma eficiente, o problema de escalonamento de tarefas em ambiente de computação heterogênea. Com o conhecimento das melhores implementações para o problema da ordenação segmentada propusemos uma nova implementação dessa heurística, cujos resultados são apresentados no próximo capítulo.

Capítulo 3

Escalonamento de Tarefas em Ambiente de Computação Heterogênea

Um sistema de computação heterogênea (HC) é composto por vários computadores com configurações diferentes, também chamados processadores ou máquinas, e um conjunto de tarefas a serem executadas por eles (Nesmachnow e Canabé, 2011). Um problema comum nesses sistemas é a forma como as tarefas a serem processadas serão distribuídas entre as máquinas, já que o tempo de execução de uma tarefa pode mudar de uma máquina para outra. Esse problema é denominado Problema de Escalonamento em Ambientes de Computação Heterogênea (HCSP - Heterogeneous Computing Scheduling Problem) (Nesmachnow e Canabé, 2011).

Apesar de muitas métricas de desempenho terem sido consideradas no problema de escalonamento (Leung, 2004), a métrica mais comum para avaliar o HCSP é o *makespan*. O *makespan* é definido como o tempo gasto a partir do momento em que a primeira tarefa começa até o momento em que a última tarefa é completada (Nesmachnow e Canabé, 2011). Logo, esse tempo é determinado pela máquina que irá terminar de processar suas tarefas por último. Então ao mesmo tempo que o escalonador tem que se preocupar em atribuir as tarefas nas máquinas que conseguem processá-las no menor tempo possível, também tem que se preocupar com o balanceamento da carga, para que uma máquina não fique executando por muito mais tempo que as outras.

Estimar o tempo de execução é uma técnica comum aplicada para modelar o tempo de execução das tarefas. Neste modelo, toda tarefa possui um

tempo estimado de computação (ETC - Expected Time to Compute) para cada máquina. Para estimar o tempo leva-se em consideração três propriedades: heterogeneidade da máquina, heterogeneidade da tarefa e consistência. A heterogeneidade da máquina avalia a variação dos tempos de execução de uma tarefa sobre as máquinas disponíveis, enquanto a heterogeneidade das tarefas avalia a variação dos tempos de execução das tarefas sobre uma única máquina. A terceira propriedade considera um cenário como consistente ou inconsistente. Quando uma máquina m_j executa qualquer tarefa t_i mais rápido que outra máquina m_k, o cenário é considerado consistente. Um cenário inconsistente é considerado quando uma máquina m_j pode ser mais rápida que uma máquina m_k ao executar algumas tarefas, porém mais lenta ao executar outras.

O escalonamento dessas tarefas pode ser realizado de dois modos: estático e dinâmico. Esses modos definem em que momento as decisões de escalonamento serão feitas. No caso de escalonamento estático, as informações sobre todos os recursos no HC bem como de todas as tarefas estão disponíveis no instante em que a aplicação está sendo escalonada. No caso do escalonamento dinâmico, a ideia básica é alocar tarefas à medida que elas chegam para executar (*on the fly*). Isso é útil quando é impossível determinar o tempo de execução, direções de desvios, número de iterações de um laço e nos casos de tarefas estarem chegando em tempo real. A vantagem sobre o escalonamento estático é que o sistema não precisa se preocupar com o comportamento do tempo de execução da aplicação antes da execução.

No modo estático, toda tarefa que compõe o trabalho é atribuída uma única vez a uma máquina. Assim, conseguimos estimar com mais precisão o custo de computação da execução atual. Um dos maiores benefícios do modelo estático é que é mais fácil de programar o escalonador, pois esse modelo permite uma visão global das tarefas e dos custos.

No modelo de tarefas independentes, todas as tarefas podem ser executadas independentemente, sem se preocupar com a ordem de execução. As aplicações a serem executadas são compostas por uma coleção de tarefas indivisíveis que não tem dependência entre si, comumente chamadas de *metatask*. Problemas de escalonamento consistem em tentar minimizar o tempo gasto para executar todas essas tarefas. A formalização abaixo apresenta o modelo matemático para o HCSP com o intuito de minimizar o *makespan*:

- Dado um sistema de computação heterogênea composto de um conjunto de M máquinas $P = m_1, m_2, ..., m_M$ e uma coleção de N tarefas $T = t_1, t_2, .., t_N$ a serem executadas.

- Considere uma função de tempo de execução $ET : PxT \to R^+$, em que $ET(t_i, m_j)$ é o tempo necessário para executar a tarefa t_i na máquina m_j.

- O objetivo do HCSP é encontrar uma atribuição de tarefas às máquinas (uma função $f: T^N \rightarrow P^M$) que minimize o *makespan*, conforme definido na Equação 3.1.

$$\max_{m_j \in P} \sum_{\substack{t_i \in T \\ f(t_i)=m_j}} ET(t_i, m_j) \tag{3.1}$$

3.1 Trabalhos Relacionados

Como o problema do escalonamento de tarefas em ambiente de computação heterogênea é um problema NP-completo (Ullman, 1975), um grande número de heurísticas foram propostas para obter soluções semi-ótimas do problema (Maheswaran et al., 1999). Nesta seção, apresentamos trabalhos anteriores que efetuaram a comparação de heurísticas para resolver esse problema.

Min-You Wu et al. (2000), propuseram a heurística *segmented min-min*, que é uma alteração na heurística min-min para reduzir o desbalanceamento de carga. Eles ordenam as tarefas em ordem decrescente dos tempos e dividem em vários segmentos que são escalonados em ordem. Ao aumentar a quantidade de segmentos o desbalanceamento de carga é reduzido. Por outro lado, ao aumentar muito o número de segmentos, as vantagens da heurística min-min são perdidas. Em virtude disso, eles fixaram o número de segmentos em quatro e obtiveram resultados melhores que o algoritmo min-min tradicional.

Wu e Shu (2001) apresentaram um algoritmo que reduz o desbalanceamento de carga produzido pela heurística min-min. Como o algoritmo min-min dá maior prioridade às tarefas menores, as tarefas maiores são escalonadas no final, o que acaba gerando o desbalanceamento de carga. Nesse trabalho, eles criam o algoritmo *Relative Cost*, uma solução em que o mapeamento das tarefas maiores é realizado antes das tarefas menores. Com isso, conseguiram melhorar o balanceamento de carga e obter resultados melhores que o max-min e min-min em um tempo similar. Os resultados também mostraram que algoritmos genéticos (GA) conseguem melhorar o mapeamento já existente, porém demandam bem mais tempo.

Braun et al. (2001) desenvolveram experimentos sobre onze heurísticas diferentes. Em seus testes, utilizaram cenários com características de heterogeneidade consistente e inconsistente, bem como heterogeneidade de máquina alta e baixa. Os resultados mostraram que essas características afetam diretamente a qualidade da solução gerada por essas heurísticas. Além disso, pode-se evidenciar também que os algoritmos genéticos (GA) tiveram os melhores resultados na maioria dos cenários, mas que o min-min pode ser resolvido

muito rapidamente com resultados próximos ao GA.

Etminani e Naghibzadeh (2007) desenvolveram um algoritmo que seleciona a melhor solução entre as heurísticas min-min e max-min em cada iteração do algoritmo, baseado na utilização das máquinas e no tamanho das tarefas que ainda precisam ser escalonadas. Os resultados mostraram que essa heurística é melhor que as heurística min-min e max-min, quando aplicadas isoladamente. O novo algoritmo reduziu o *makespan*, aumentou a utilização das máquinas e o balanceamento de carga.

Nesmachnow et al. (2010) apresentaram e avaliaram algoritmos evolucionários (EAs) sequenciais e paralelos usando a biblioteca de otimização combinatória MALLBA. Os algoritmos paralelos dividem a população original em subpopulações nas quais são executadas EAs sequenciais. Ocasionalmente indivíduos são trocados entre as subpopulações. Obtiveram resultados superiores ao min-min e aos GAs tradicionais.

Kokilavani et al. (2011) propuseram o *Load Balanced Min-Min* (LBMM). Uma adaptação da heurística min-min para melhorar o balanceamento de carga. Após aplicar o min-min, percorrem-se as máquinas analisando a carga de cada uma e fazendo o reescalonamento quando necessário. Para isso, encontra a máquina com o maior *makespan* e a tarefa nela que possui o menor tempo esperado. Depois procura qual máquina irá gerar o maior tempo de conclusão para essa tarefa. Se o maior tempo de conclusão for menor que o *makespan*, reescalona essa tarefa na nova máquina e calcula o novo tempo. Executa esses passos até que todas as máquinas tenham sido consideradas.

Li et al. (2011) propuseram uma política de escalonamento de tarefas baseada no *Ant Colony Optimization* (ACO). Com esse algoritmo pretendiam melhorar o balanceamento de carga entre as máquinas virtuais, ao mesmo tempo que tentavam minimizar o makespan. O experimentos realizados no CloudSim, mostraram que o algoritmo proposto (LBACO) consegue ser melhor que o *First Come First Serve* (FCFS) e o *Ant Colony Optimization* (ACO) tradicional.

Nesmachnow e Canabé (2011) apresentaram uma implementação em GPU para as heurísticas min-min e *sufferage*. As duas implementações efetuam a paralelização por tarefas, ou seja, cada tarefa é atribuída a uma *thread* que será responsável por encontrar a máquina que executa essa tarefa no menor tempo. Então, uma redução paralela é efetuada para selecionar qual tarefa será atribuída a qual máquina. Com essas implementações mostraram que utilizando GPU podemos atingir *speedups* de aproximadamente seis vezes em relação a uma implementação sequencial.

Kumar e Verma (2012) mostraram que ao utilizar um algoritmo genético, a escolha da população inicial é de fundamental importância. Fizeram a comparação entre o algoritmo genético padrão, que utiliza uma população inicial

gerada aleatoriamente, e um algoritmo genético modificado, que utiliza como população inicial as heurísticas max-min e min-min. Eles utilizaram o CloudSim para validar o algoritmo e concluíram que um algoritmo genético consegue resultados melhores quando a sua população inicial já é uma boa solução para o problema sendo resolvido.

Canabé e Nesmachnow (2012) propuseram quatro implementações utilizando GPU para a heurística min-min. A primeira usa a mesma estratégia de paralelização da implementação de 2011 Nesmachnow e Canabé (2011). Outras duas estratégias que dividem o domínio em quatro subgrupos e aplica o min-min em cada grupo independentemente. Ambos utilizam implementações híbridas CPU-GPU, uma com *Pthreads* e outra com OMP. A quarta implementação também divide o domínio em quatro subgrupos, porém a cada iteração escolhe apenas a melhor tarefa entre as quatro soluções geradas e escalona em uma máquina. É importante observar que essas implementações são variações do min-min, pois fornecem soluções aproximadas.

Ezzatti et al. (2013) fizeram um ajuste no algoritmo da heurística min-min para resolvê-la com complexidade $O(tm)$, enquanto o algoritmo tradicional possui complexidade $O(t^2m)$ com t sendo o número de tarefas e m o número de máquinas. O algoritmo efetua uma ordenação segmentada nas linhas da matriz máquinas-tarefas para colocar as tarefas que executam mais rápido em cada máquina nas primeiras colunas. Dessa forma, o próximo estágio consiste em avaliar os valores das tarefas na ordem em que estão em cada linha, tornando desnecessário varrer toda a matriz a cada iteração. Uma implementação sequencial desse algoritmo foi comparada com implementações em GPU do algoritmo clássico, a nova implementação mostrou-se mais eficiente que as implementações em GPU.

Bhoi e Ramanuj (2013) fizeram uma alteração no algoritmo *improved max-min* para, no lugar de selecionar as tarefas maiores para serem escalonadas primeiro, escolher as tarefas que possuem a média do tempo de execução das tarefas que faltam escalonar ou que estão um pouco acima da média. A tarefa selecionada por esses dois algoritmos em cada passo é atribuída à máquina que possui o menor tempo de conclusão. A esse algoritmo deram o nome de *enhanced max-min* e conseguiram mostrar, por meio de testes no simulador CloudSim, que essa variação no algoritmo proporciona um *makespan* menor.

Agarwal et al. (2014) apresentaram um algoritmo que dá prioridade de execução às tarefas maiores e às máquinas mais rápidas. A cada iteração a máquina mais rápida que estiver disponível é escolhida para executar a tarefa mais curta que ainda não foi escalonada. Eles simularam um ambiente de computação em nuvem utilizando o CloudSim e o algoritmo proposto (*GPA - General Prority Algorithm)* mostrou-se superior às heurísticas *First Come First*

Served (FCFS) e *Roud Robin (RR)*.

Tsai et al. (2014) propuseram o *Hyper Heuristic Scheduling Algorithm (HHSA)*. Um algoritmo que seleciona aleatoriamente uma heurística, a partir do conjunto de heurísticas disponíveis e executa o número de iterações que satisfaz seus critérios. A cada execução são verificados se critérios de modificação e melhorias estão sendo satisfeitos para aquela heurística. Caso contrário, seleciona aleatoriamente uma nova. Os resultados mostraram que esse método consegue convergir mais rapidamente que os outros testados e apresentou os melhores resultados para a maioria dos casos.

Patel et al. (2015) apresentaram uma variação do *Load Balanced Min-Min* (LBMM) para melhorar o balanceamento de carga. A diferença para o LBMM é que o ELBMM, no lugar de selecionar a tarefa com menor ETC, seleciona a tarefa com maior ETC na máquina que termina de executar por último (*makespan*). Os outros passos do algoritmo são iguais ao LBMM. Seus testes mostraram que a heurística ELBMM pode ser melhor que o min-min e LBMM.

Pedemonte et al. (2016) fizeram uma implementação híbrida GPU-CPU do *min-min sort*. A primeira parte do algoritmo, que consiste em ordenar as linhas da matriz, foi realizada pela GPU, utilizando a biblioteca ModernGPU (MGPU), enquanto as próximas iterações, que consistem de identificar o par tarefa máquina que gera o menor *makespan* ocorrem na CPU. Além disso, duas implementações paralelas em CPU foram implementadas, a primeira usando o Matlab e a segunda usando C++ com OpenMP. A implementação em C++ com openMP apresentou os melhores resultados para dimensões pequenas do problema, enquanto que a versão híbrida provou-se mais eficiente para dimensões grandes. Porém, para resolver a primeira etapa, ordenação das linhas da matriz, a GPU provou-se mais eficiente em todos os casos.

Rajput e Kushwah (2016) propuseram o *Improved Load Balanced Min-Min* (ILBMM) para escalonar tarefas em um ambiente de computação em nuvem. O ILBMM é uma melhoria no LBMM, que aplica um algoritmo genético para tentar melhorar a solução. Já o LBMM é uma adaptação da heurística min-min para melhorar o balanceamento de carga, que tenta um reescalonamento das tarefas após aplicar o min-min. Seus resultados produziram um *makespan* melhor e aumentaram a utilização dos recursos, quando comparado ao LBMM.

Massobrio et al. (2018) utilizaram o *Virtual Savant* (VS) para gerar soluções eficientes para o HCSP. O VS é treinado usando *Support Vector Machines* (SVMs) para aprender automaticamente como resolver problemas de otimização a partir de um conjunto de observações, obtidos de um algoritmo de referência. O algoritmo de referência utilizado por eles foi o *min-min*. O VS consiste em duas fases: classificação, etapa em que os resultados de ins-

tâncias desconhecidas do problema são preditas, e otimização, etapa em nos quais os resultados preditos são otimizados usando rotinas de busca específicos. A etapa de classificação recebe como entrada a matriz com os tempos de execução das tarefas em cada máquina e, utilizando *Support Vector Machines* (SVMs), cria uma matriz que armazena, para cada tarefa, a probabilidade de ser escalonada em cada máquina. Utilizando essa matriz de probabilidades, a etapa de otimização gera aleatoriamente um conjunto de soluções para o problema, e aplica uma busca local em cada uma delas. A busca local consiste em iterativamente mover uma tarefa escolhida aleatoriamente da máquina com o maior tempo de conclusão para uma das N máquinas com o menor tempo de conclusão. A escolha da máquina é feita considerando a máquina com o menor tempo de conclusão, após a tarefa ser movida para ela. Quando todas as buscas locais são concluídas, a melhor solução encontrada é retornada. Os autores executaram o algoritmo em uma Xeon Phi e mostraram que para uma solução mais acurada, que reduza melhor o *makespan*, é indicado a utilização de mais *threads*. Além disso, compararam os resultados com o min-min e conseguiram, em uma das entradas do problema, um *makespan* 15% melhor que o min-min.

Como trata-se da heurística mais rápida para escalonamento estático de tarefas e apresenta bons resultados, a maioria dos trabalhos citados acabam efetuando a comparação de seus resultados também com a heurística min-min. Por isso, definimos o min-min como a base para nossos estudos.

3.2 Soluções Propostas

Existem duas abordagens para a heurística min-min na literatura, a heurística *min-min clássica* (Ibarra e Kim, 1977) e a heurística proposta por Ezzatti et al. (2013), que chamamos nesse trabalho de *min-min sort*. A primeira percorre todas as posições da matriz tarefas-máquinas a cada iteração, procurando pelo par tarefa-máquina que gera o menor tempo de conclusão. A segunda otimiza o algoritmo efetuando uma ordenação segmentada da matriz antes de iniciar as iterações de busca pelos pares tarefas-máquinas.

Desenvolvemos duas implementações para o *min-min clássico*, uma sequencial em CPU e uma paralela em GPU. A versão paralela desse algoritmo mostrou-se mais rápida que a versão sequencial na maioria dos testes realizados, sendo a melhor opção para grandes dimensões do problema.

Pedemonte et al. (2016) apresentaram uma implementação híbrida GPU-CPU do *min-min sort* com resultados competitivos quando comparados a outras implementações paralelas. A versão dos autores utiliza uma implementação baseada no *merge sort* para ordenar as linhas da matriz, resultando em

um algoritmo de complexidade $O(mt\log t)$. Visando um algoritmo de complexidade assintótica linear $O(mt)$, usamos uma implementação baseada no *radix sort* para realizar a ordenação segmentada, e criamos uma implementação híbrida GPU-CPU e uma implementação totalmente em GPU. Considerando que o código dos autores não foi disponibilizado, uma implementação híbrida GPU-CPU e uma implementação em GPU baseadas no *merge sort* também foram implementadas para fins de comparação. Além das implementações que utilizam GPU, também foi implementada uma versão sequencial para o *min-min sort*. As versões híbridas foram competitivas entre si e apresentaram os melhores resultados para dimensões pequenas do problema, enquanto que as implementações em GPU mostraram-se eficientes para dimensões grandes do problema, e também foram competitivas entre si.

3.2.1 Min-min clássico

O algoritmo *min-min clássico* percorre toda a matriz tarefas-máquinas a cada iteração do algoritmo para encontrar o par que gera o menor tempo de conclusão. A cada iteração um vetor com o tempo de conclusão de cada máquina é atualizado de acordo com as tarefas atribuídas a ela. Essa atualização ocorre considerando o tempo que cada tarefa ainda não escalonada acrescentará àquela máquina. A tarefa que se acrescentada a uma máquina acarrete o menor tempo de conclusão será a escolhida para ser escalonada naquela iteração. O tempo dessa tarefa é acrescentado à respectiva máquina e a tarefa é marcada como já escalonada. Esses passos estão descritos no Algoritmo 7, cuja complexidade assintótica é $O(mt^2)$.

Algoritmo 7 min-min clássico

```
1:  function MIN-MIN-PADRAO(tasks, completion, t, m)
2:      for j = 1 to m do
3:          completion[j] = 0
4:      for k = 1 to t do
5:          min = ∞
6:          for i = 1 to t do
7:              if task i not mapped then
8:                  for j = 1 to m do
9:                      if tasks[i][j] + completion[j] < min then
10:                         min = tasks[i][j] + completion[j]
11:                         imin = i
12:                         jmin = j
13:         completion[jmin] = min
14:         Set task imin as mapped
```

Na matriz apresentada no Algoritmo 7 as tarefas estão representadas pelas linhas e as máquinas pelas colunas da matriz. Porém as implementações

em GPU, bem como a implementação sequencial do algoritmo *min-min sort*, armazenam a matriz tarefas-máquinas com as máquinas sendo representadas pelas linhas e as tarefas sendo representadas pelas colunas. Dessa forma, optamos por apresentar o exemplo da Tabela 3.1 usando a estrutura utilizada pela maior parte das implementações, e também para facilitar o entendimento da diferença entre os algoritmos *min-min clássico* e *min-min sort*.

A Tabela 3.1 apresenta um exemplo de execução do algoritmo em um ambiente com 3 máquinas e 5 tarefas. O passo 0 é o início do algoritmo, ainda não foram atribuídas tarefas a nenhuma máquina. Toda a matriz é percorrida em cada passo do algoritmo. No passo 1 a tarefa T5 foi selecionada para ser escalonada na máquina M3, pois é o par tarefa-máquina que gera o menor tempo de conclusão. O tempo de conclusão da máquina M3 é atualizado para 11, que é o tempo estimado de execução da tarefa T5 na máquina M3, e a tarefa T5 é marcada como já escalonada (denotada na tabela pelo texto tachado). No passo 2, o par tarefa-máquina que gera o menor tempo de conclusão é a tarefa T2 sendo executada na máquina M1, então o tempo de conclusão da máquina M1 é atualizado para 12, e a tarefa T2 é marcada como já escalonada. Mesmo a máquina M2 ainda não tendo tarefas escalonadas, no passo 3, a tarefa T1 é selecionada para executar na máquina M3, e, no passo 4, a tarefa T4 é escalonada na máquina M1, só depois, no passo 5, a máquina M2 receberá a tarefa T3 para executar. No passo 6 o *makespan* das tarefas está destacado e corresponde ao maior tempo de conclusão entre todas as máquinas. No exemplo, o *makespan* é igual a 31.

A implementação em GPU da heurística min-min clássica utiliza as linhas da matriz para representar as máquinas e as colunas para representar as tarefas. O algoritmo paralelo consiste basicamente em realizar em cada iteração várias reduções paralelas nas linhas da matriz, para encontrar o par tarefa-máquina que gera o menor tempo de conclusão. A estratégia de paralelização empregada nesse trabalho consiste em atribuir a cada *thread* CUDA a responsabilidade de processar uma tarefa, ou seja, encontrar a máquina em que a tarefa possui o menor tempo de conclusão. Os resultados encontrados por cada *thread* são inicialmente armazenados em um vetor na memória compartilhada, que depois de uma redução paralela local, entre as *threads* do bloco CUDA, é realizada uma redução paralela global, entre os blocos CUDA. Ao final das reduções a tarefa que deve ser escalonada naquela iteração é encontrada. Como a cada passo as *threads* estão acessando a mesma linha da matriz, como apresentado na Figura 3.1, isso favorece a coalescência e, consequentemente, o desempenho dessa estratégia.

Tabela 3.1: Exemplo de execução do algoritmo *min-min clássico* em um ambiente com 3 máquinas e 5 tarefas.

		Tarefas					
Passo	Máquinas	T1	T2	T3	T4	T5	Tempo de Conclusão
0	M1	41	12	39	17	82	M1=0
	M2	43	21	31	71	28	M2=0
	M3	14	13	93	73	11	M3=0
1	M1	41	12	39	17	82	M1=0
	M2	43	21	31	71	28	M2=0
	M3	14	13	93	73	11	M3=0+11
2	M1	41	12	39	17	~~82~~	M1=0+12
	M2	43	21	31	71	~~28~~	M2=0
	M3	14	13	93	73	~~11~~	M3=11
3	M1	41	~~12~~	39	17	~~82~~	M1=12
	M2	43	~~21~~	31	71	~~28~~	M2=0
	M3	14	~~13~~	93	73	~~11~~	M3=11+14=25
4	M1	~~41~~	~~12~~	39	17	~~82~~	M1=12+17=29
	M2	~~43~~	~~21~~	31	71	~~28~~	M2=0
	M3	~~14~~	~~13~~	93	73	~~11~~	M3=25
5	M1	~~41~~	~~12~~	39	~~17~~	~~82~~	M1=29
	M2	~~43~~	~~21~~	31	~~71~~	~~28~~	M2=0+31
	M3	~~14~~	~~13~~	93	~~73~~	~~11~~	M3=25
6	M1	~~41~~	~~12~~	~~39~~	~~17~~	~~82~~	M1=29
	M2	~~43~~	~~21~~	~~31~~	~~71~~	~~28~~	**M2=31**
	M3	~~14~~	~~13~~	~~93~~	~~73~~	~~11~~	M3=25

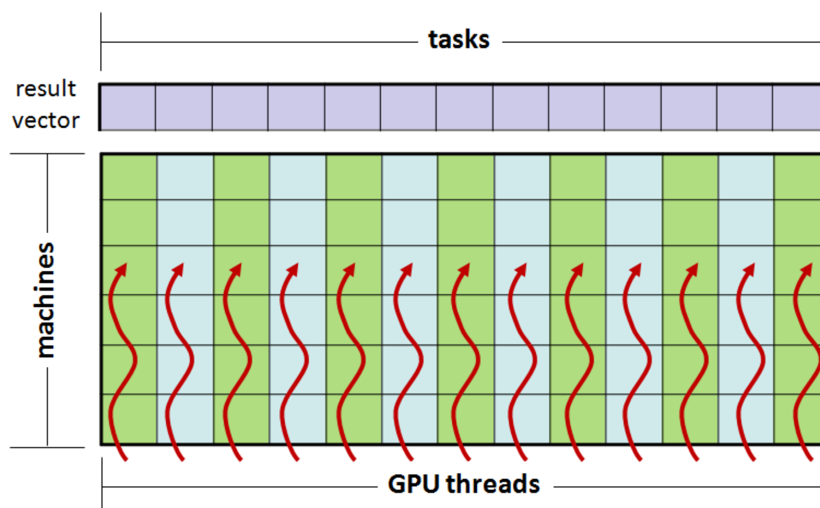


Figura 3.1: Estratégia de implementação em GPU da heurística min-min (Nesmachnow e Canabé, 2011).

3.2.2 Min-Min Sort

O algoritmo *min-min sort* (Ezzatti et al., 2013) otimiza a tarefa de seleção em cada iteração ao efetuar uma ordenação segmentada da matriz antes de iniciar a busca pela tarefa. Esse passo estabelece a ordem em que os elementos de cada máquina serão analisados a cada iteração, não precisando mais percorrer toda a matriz para encontrar o par tarefa-máquinas que gera o menor tempo de conclusão. Dessa forma, a complexidade assintótica é reduzida de $O(mt^2)$ para $O(mt \log t)$. Os passos dessa estratégia são apresentados no Algoritmo 8. Utilizando um algoritmo de ordenação linear no passo 4, é possível reduzir a complexidade assintótica desse algoritmo para $O(mt)$.

Algoritmo 8 Min-Min Sort

```
 1: function MIN-MIN-SORT(tasks, completion, t, m)
 2:     for j = 1 to m do
 3:         completion[j] = 0
 4:     SEGMENTED − SORTING(tasks)                    ▷ Sort matrix rows
 5:     for k = 1 to t do
 6:         min = ∞
 7:         for j = 1 to m do
 8:             i = machine current task not mapped
 9:             if tasks[j][i] + completion[j] < min then
10:                 min = tasks[j][i] + completion[j]
11:                 jmin = j
12:                 imin = i
13:         completion[jmin] = min
14:         Set task imin as mapped
```

A Tabela 3.2 apresenta um exemplo de execução do algoritmo em um ambiente com 3 máquinas e 5 tarefas. O passo 0 é o início do algoritmo, ainda não foi realizada nenhuma etapa. No passo 1, é realizada a ordenação segmentada da matriz, isto é, as tarefas são colocadas em ordem ascendente de tempo de execução em cada máquina. No passo 2 é iniciada a seleção das tarefas, note que essa seleção não precisa mais ser realizada na matriz inteira, nesta iteração, só a primeira coluna da matriz precisa ser avaliada, que é onde se identifica que a tarefa T5 sendo escalonada na máquina M3 é a que gera o menor tempo de conclusão. O tempo de conclusão da máquina M3 é atualizado para 11, e a tarefa T5 é marcada como já escalonada (na tabela denotado pelo texto tachado). No passo 3, a primeira coluna de M1 e M2 serão avaliadas para seleção, bem como a segunda coluna de M3. Note que essa avaliação é sempre da esquerda para direita levando-se em conta somente tarefas que ainda não foram escalonadas. O par tarefa-máquina selecionado nessa iteração é a tarefa T2 sendo executada na máquina M1, então o tempo de conclusão da máquina M1 é atualizado para 12, e a tarefa T2 é marcada

Tabela 3.2: Exemplo do algoritmo *min-min sort* em um ambiente com 3 máquinas e 5 tarefas.

Passo	Máquinas	Tarefas					Tempo de Conclusão
0	M1	41(T1)	12(T2)	39(T3)	17(T4)	82(T5)	M1=0
	M2	43(T1)	21(T2)	31(T3)	71(T4)	28(T5)	M2=0
	M3	14(T1)	13(T2)	93(T3)	73(T4)	11(T5)	M3=0
1	M1	12(T2)	17(T4)	39(T3)	41(T1)	82(T5)	M1=0
	M2	21(T2)	28(T5)	31(T3)	43(T1)	71(T4)	M2=0
	M3	11(T5)	13(T2)	14(T1)	73(T4)	93(T3)	M3=0
2	M1	12(T2)	17(T4)	39(T3)	41(T1)	82(T5)	M1=0
	M2	21(T2)	28(T5)	31(T3)	43(T1)	71(T4)	M2=0
	M3	11(T5)	13(T2)	14(T1)	73(T4)	93(T3)	M3=0+11
3	M1	12(T2)	17(T4)	39(T3)	41(T1)	~~82(T5)~~	M1=0+12
	M2	21(T2)	~~28(T5)~~	31(T3)	43(T1)	71(T4)	M2=0
	M3	~~11(T5)~~	13(T2)	14(T1)	73(T4)	93(T3)	M3=11
4	M1	~~12(T2)~~	17(T4)	39(T3)	41(T1)	~~82(T5)~~	M1=12
	M2	~~21(T2)~~	~~28(T5)~~	31(T3)	43(T1)	71(T4)	M2=0
	M3	~~11(T5)~~	~~13(T2)~~	14(T1)	73(T4)	93(T3)	M3=11+14
5	M1	~~12(T2)~~	17(T4)	39(T3)	~~41(T1)~~	~~82(T5)~~	M1=12+17
	M2	~~21(T2)~~	~~28(T5)~~	31(T3)	~~43(T1)~~	71(T4)	M2=0
	M3	~~11(T5)~~	~~13(T2)~~	~~14(T1)~~	73(T4)	93(T3)	M3=26
6	M1	~~12(T2)~~	~~17(T4)~~	39(T3)	~~41(T1)~~	~~82(T5)~~	M1=29
	M2	~~21(T2)~~	~~28(T5)~~	31(T3)	~~43(T1)~~	~~71(T4)~~	M2=0+31
	M3	~~11(T5)~~	~~13(T2)~~	~~14(T1)~~	~~73(T4)~~	93(T3)	M3=26
7	M1	~~12(T2)~~	~~17(T4)~~	~~39(T3)~~	~~41(T1)~~	~~82(T5)~~	M1=29
	M2	~~21(T2)~~	~~28(T5)~~	~~31(T3)~~	~~43(T1)~~	~~71(T4)~~	**M2=31**
	M3	~~11(T5)~~	~~13(T2)~~	~~14(T1)~~	~~73(T4)~~	~~93(T3)~~	M3=26

como já escalonada. No passo 4, a tarefa T1 é selecionada para executar na máquina M3, pois é o par que gera o menor tempo de conclusão, e a tarefa T4 será escalonada no passo 5 para ser executada na máquina M1. Só depois, no passo 6, a máquina M2 receberá a primeira tarefa para executar. No passo 7 o *makespan* das tarefas está destacado e corresponde ao maior tempo de conclusão entre todas as máquinas.

As implementações em GPU da heurística *min-min sort* efetuam a ordenação das linhas da matriz utilizando uma biblioteca CUDA, e depois são realizadas reduções paralelas para identificar as máquinas que possuem o menor tempo de conclusão em cada iteração do algoritmo. Cada *thread* CUDA é responsável por processar uma máquina, isto é, copiar para a memória compartilhada o tempo de execução da menor tarefa naquela máquina. Uma redução paralela local, entre as *threads* do bloco CUDA, é realizada e depois uma redução paralela global, entre os blocos CUDA. Ao final das reduções a tarefa que deve ser escalonada naquela iteração é encontrada. A diferença para o

algoritmo paralelo do *min-min clássico* é que, em cada iteração, as *threads* tinham que percorrer toda a linha da matriz, enquanto que neste somente até encontrar uma tarefa ainda não escalonada.

Foram testadas duas bibliotecas de ordenação segmentada: MGPU e CUB. A primeira está presente na implementação de Ezzatti et al. (2013) para esse problema e é baseada no *merge sort*, e a segunda é baseada no radix sort e torna a solução do problema linear. As versões híbridas CPU-GPU do min-min sort resolvem a ordenação segmentada da matriz em GPU usando os mesmos passos e bibliotecas da versão paralela. Após a ordenação segmentada, os dados são copiados de volta para CPU e o mesmo algoritmo da versão sequencial do *min-min sort* é executado.

3.3 Resultados Experimentais

Os experimentos deste capítulo foram realizados em uma máquina com 32 GB de memória RAM, processador Intel(R) Xeon(R) CPU E5-1620 v3 @ 3.50GHz e placa gráfica Quadro M4000 com 8 GB de memória, sistema operacional Ubuntu 16.04, g++ versão 4.9 e CUDA versão 10.1. Os cenários de teste foram gerados usando o gerador disponível publicamente em http://www.fing.edu.uy/inco/grupos/cecal/hpc/HCSP.

Foram comparadas, nesta seção, as implementações: sequencial da heurística min-min clássica (**P**); paralela da heurística min-min clássica (**GP**); sequencial da heurística *min-min sort* (**S**); paralela da heurística *min-min sort* usando a biblioteca CUB (**GC**); paralela da heurística *min-min sort* usando a biblioteca ModernGPU (**GM**); híbrida da heurística *min-min sort* usando a biblioteca CUB (**HC**); híbrida da heurística *min-min sort* usando a biblioteca ModernGPU (**HM**).

As dimensões testadas para o problema foram: 1024, 2048, 8192, 16384, 32768 e 65536 tarefas, que precisavam ser distribuídas em 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096 e 8192 máquinas. Os testes só foram rodados para o par de dimensões tarefas-máquinas maior ou igual a 2, isto é, a proporção de pelo menos 2 tarefas por máquinas. O limite superior dos testes se deu pelo limite da memória global da GPU, enquanto que o limite inferior foi definido para que tenhamos quantidade de tarefas suficientemente grandes que faça sentido testá-las em GPU.

A Figura 3.2 mostra o comportamento das implementações ao aumentar o número de máquinas, enquanto o número de tarefas permanece constante em 32768. As implementações que usam a CPU para selecionar as tarefas em cada iteração sofrem impacto semelhante entre si, enquanto que implementações que usam a GPU na seleção das tarefas, sofrem bem menos impacto com

esse aumento. Isso indicou que para dimensões maiores do problema a GPU era mais indicada para uma solução eficiente.

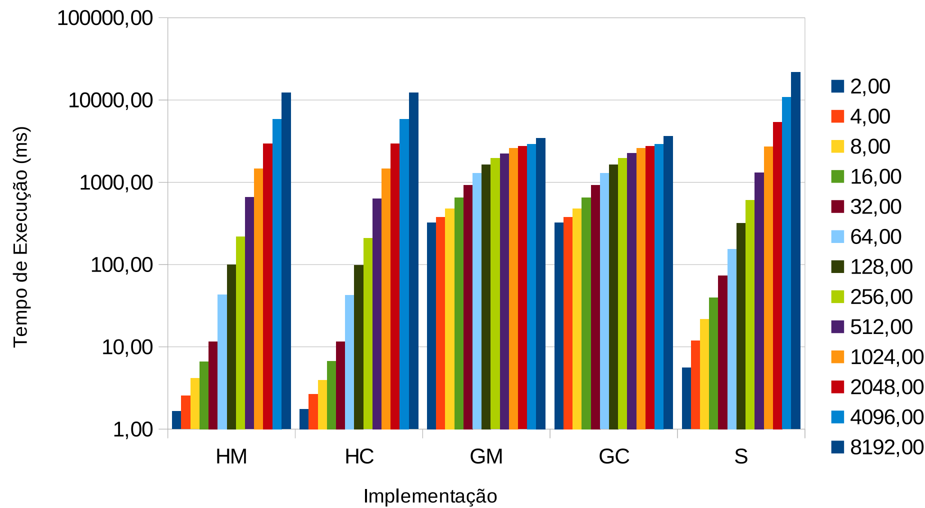


Figura 3.2: Comportamento das implementações ao aumentar o número de máquinas, enquanto mantém a quantidade de tarefas em 32768.

A Figura 3.3 mostra o comportamento das implementações ao aumentar o número de tarefas, enquanto o número de máquinas se mantém constante em 512. Observe que o comportamento de todas as implementações são muito semelhantes, apesar de cada uma ter um tempo de execução diferente. Isso sugere que o número de tarefas sendo escalonadas não é o fator preponderante na comparação entre as implementações, o que não acontece com o aumento no número de máquinas.

A Figura 3.4 apresenta os tempos de execução de cada implementação para escalonar 32768 tarefas nas quantidades de máquinas indicadas. À medida que o número de máquinas aumenta as implementações híbridas e sequencial são afetadas na mesma proporção, enquanto que nas implementações em GPU o impacto é bem menor. O que indica que para dimensões grandes do problema a GPU pode ser uma boa solução.

Pedemonte et al. (2016) propuseram uma implementação híbrida GPU-CPU para o *min-min sort* e compararam com versões paralelas em CPU. Os resultados mostraram que, para dimensões grandes do problema, a versão híbrida era a mais indicada. Utilizamos as dimensões testadas por eles para comparar nossas implementações, e criamos a Figura 3.5.

A Figura 3.5 está em escala logarítmica e apresenta os *speedups* de todas as implementações quando comparadas com a versão sequencial clássica do min-min. As implementações baseadas no *min-min sort* são mais eficientes para todas as dimensões do problema que as implementações baseadas no *min-min clássico*, podendo atingir *speedup* de até 400 vezes com uma implementação eficiente. A figura também sugere que o aumento nas dimensões do

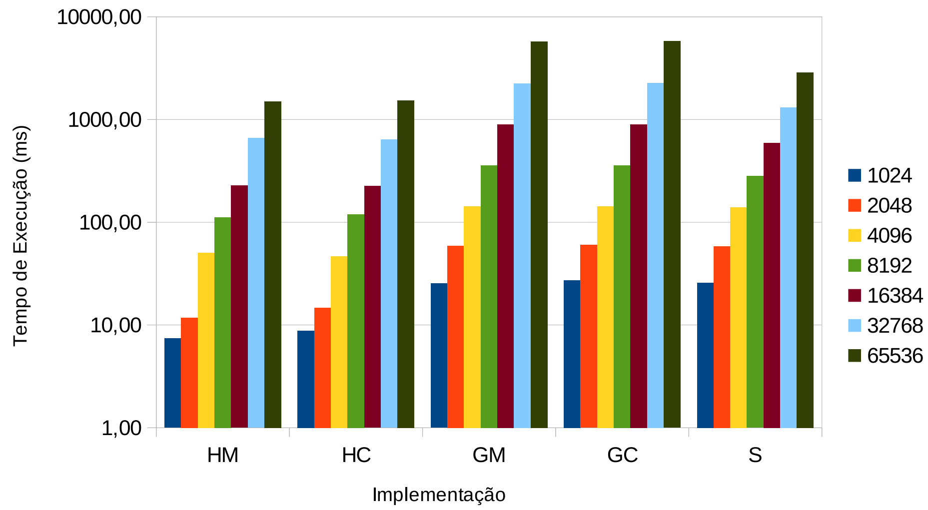


Figura 3.3: Comportamento das implementações ao aumentar o número de tarefas, enquanto mantém a quantidade de máquinas em 512.

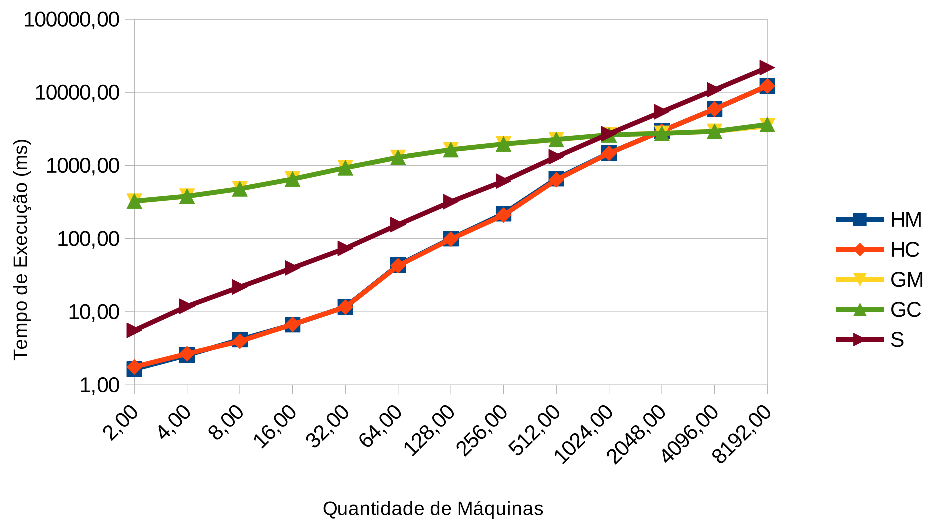


Figura 3.4: Tempo de execução das implementações do *min-min sort*.

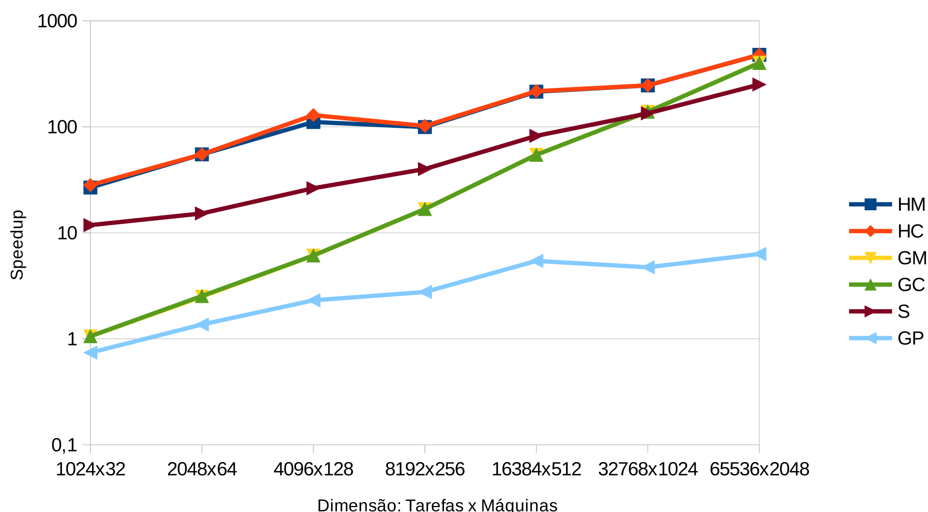


Figura 3.5: *Speedup* das implementações do min-min em relação à implementação sequencial padrão.

problema favorece as implementações do *min-min sort* e as implementações em GPU.

A Tabela 3.3 apresenta os tempos de execução das implementações do *min-min clássico*, bem como os *speedups* para cada dimensão do problema. Ao avaliar os *speedups*, conseguimos atingir desempenhos até seis vezes melhores em GPU que em CPU, indicando que a GPU é mais indicada para resolver de forma eficiente esse algoritmo.

Tabela 3.3: Tempos de execução das implementações do *min-min clássico* e os respectivos *speedups* quando comparadas com a versão sequencial.

Tarefas x Máquinas	**GP**	**P**	*speedup*
1024x32	27,02	19,99	0,74
2048x64	88,04	120,01	1,36
4096x128	334,97	772,20	2,31
8192x256	1999,22	5523,82	2,76
16384x512	8991,11	48850,75	5,43
32768x1024	76732,80	363306,97	4,73
65536x2048	456937,40	2886585,00	6,32

A Tabela 3.4 apresenta os *speedups* das implementações do *min-min sort* em GPU (**GC** e **GM**) e as versões híbridas (**HC** e **HM**), quando comparadas com a implementação sequencial. As versões híbridas foram as melhores em todas as dimensões apresentadas na tabela, enquanto que as versões em GPU são melhores que a versão sequencial somente para entradas grandes do problema, como nas dimensões 3278x1024 e 65536x2048.

As versões híbridas (**HC** e **HM**) possuem *speedups* melhores nas primeiras dimensões, podendo ser mais de 4 vezes superiores que a versão sequencial. Mas essa vantagem reduz com o aumento da dimensão do problema. Isso é

Tabela 3.4: Tempos de execução das implementações do *min-min sort* e o *speedup* das mesmas, quando comparadas com a versão sequencial.

		HM		**HC**		**GM**		**GC**	
Tarefas x Máquinas	**S**	Tempo	*Speedup*	Tempo	*Speedup*	Tempo	*Speedup*	Tempo	*Speedup*
1024x32	1,69	0,75	2,26	0,71	2,39	18,98	0,09	18,93	0,09
2048x64	7,87	2,18	3,61	2,18	3,61	48,17	0,16	47,43	0,17
4096x128	29,32	6,95	4,22	5,99	4,90	126,67	0,23	126,37	0,23
8192x256	138,15	55,51	2,49	54,19	2,55	330,20	0,42	329,33	0,42
16384x512	594,15	227,68	2,61	225,21	2,64	896,51	0,66	896,18	0,66
32768x1024	2708,82	1477,63	1,83	1474,03	1,84	2621,74	1,03	2615,87	1,04
65536x2048	11512,65	6041,30	1,91	6035,43	1,91	7165,62	1,61	7205,89	1,60

explicado pelas características do algoritmo *min-min sort*, que é dividido em duas partes. A primeira, consiste da ordenação segmentada da matriz, que a GPU provou-se mais eficiente, e é o motivo da versão híbrida ter os melhores tempos nessas dimensões. Já a segunda parte do algoritmo, consiste em encontrar o par tarefa-máquina que gera o menor *makespan*. Como as tarefas de cada máquina já estão ordenadas pelo tempo estimado de execução nelas, essa etapa consiste basicamente de comparar uma tarefa de cada máquina. Quando a quantidade de máquinas é pequena, essa etapa não tem tanto peso no tempo de execução, mas à medida que esse número aumenta, o impacto no tempo de execução reduz o *speedup* das implementações híbridas em relação à implementação sequencial, pois ambas possuem o mesmo código nessa etapa. Pelo mesmo motivo, as versões em GPU (**GC** e **GM**) foram melhores que a versão sequencial nas dimensões grandes do problema. Enquanto o número de máquinas é pequeno, resolver a segunda etapa do problema usando uma versão sequencial é mais interessante, porém, à medida que o número de máquinas cresce, resolvê-la em GPU, passa a ser mais eficiente.

Como os resultados apresentados por Pedemonte et al. (2016) testam apenas dimensões cuja proporção tarefas-máquinas são as mesmas, não é possível avaliar todas as possibilidades em relação às melhores implementações, já que em nossas implementações essas dimensões favoreceram as versões híbridas do algoritmo. Dessa forma, o par quantidade de máquinas e número de tarefas influencia na escolha da melhor implementação para o algoritmo.

A Tabela 3.5 apresenta para cada dimensão do problema a melhor implementação para resolvê-la. É importante notar que a melhor implementação varia não só pela quantidade de tarefas a ser escalonada, mas também pela quantidade de máquinas disponíveis. As implementações híbridas foram as melhores na maioria dos casos, alternando entre os casos em que a biblioteca MGPU (**HM**) era melhor e casos em que a biblioteca CUB (**HC**) era melhor. Porém, as implementações totalmente em GPU (**GC** e **GM**) apresentaram-se como as mais eficientes para dimensões grandes do problema.

Quando o número de máquinas cresce, as implementações em GPU são

Tabela 3.5: Melhor implementação para cada dimensão do problema rodando na Quadro M4000.

	Quantidade de Tarefas						
Qtd. Máquinas	1024	2048	4096	8192	16384	32768	65536
2	S	HC	HC	HC	HM	HM	HM
4	S	HC	HC	HC	HC	HM	HM
8	HC	HC	HC	HC	HC	HC	HM
16	HC	HC	HC	HC	HC	HM	HC
32	HC	HC	HC	HC	HC	HC	HM
64	HC	HC	HC	HC	HC	HC	HC
128	HM	HC	HC	HC	HC	HC	HM
256	HM	HM	HC	HC	HC	HC	HC
512	HM	HM	HC	HM	HC	HC	HM
1024	–	HM	HM	HC	HC	HC	HC
2048	–	–	GM	GM	GM	GC	HC
4096	–	–	–	GM	GM	GC	GM
8192	–	–	–	–	GM	GM	–

favorecidas, pois o tamanho do problema é importante para eficiência desses dispositivos de hardware. O que acontece é que na primeira parte do algoritmo *min-sort* as linhas da matriz são ordenadas, e a GPU provou-se mais eficiente nessa tarefa, favorecendo as implementações **GC**, **GM**, **HC** e **HM**. Porém, depois de realizar a ordenação, o trabalho que resta é percorrer cada máquina identificando a menor tarefa a ser escalonada. Quando o número de máquinas é pequeno, a implementação em CPU é favorecida, então as implementações híbridas acabam sendo as melhores, pois usam a GPU na ordenação segmentada e a CPU na seleção das tarefas. Porém, ao aumentar o número de máquinas, a GPU passa a ser interessante também para resolver a segunda parte do algoritmo, conforme apresentado na Tabela 3.5.

3.4 Conclusões e Trabalhos Futuros

As implementações paralelas usando GPU obtiveram resultados superiores às implementações puramente sequenciais, o que sugere que implementações paralelas são mais eficientes para resolver a heurística min-min. Além disso, os resultados mostraram que o número de máquinas pode influenciar na escolha do melhor algoritmo.

O algoritmo *min-min sort* consegue otimizar os passos do *min-min clássico*, já que se mostrou mais eficiente em todas as dimensões do problema. Enquanto a versão clássica considera toda a matriz tarefas-máquinas em cada iteração do algoritmo, a versão otimizada é divida em duas etapas: a primeira, que consiste na ordenação das linhas da matriz e possui alto grau de paralelização, tendo em vista a independência entre as linhas da matriz e a segunda

etapa, que itera somente sobre o número de máquinas em cada iteração, e, por isso, é interessante de ser resolvida em GPU quando o número de máquinas é suficientemente grande. Como nos testes realizados a dimensão de máquinas do problema não é tão grande quanto a dimensão de tarefas, a implementação híbrida da heurística obteve os melhores resultados na maioria dos casos.

Como trabalho futuro, pretendemos testar variações da heurística min-min, como os algoritmos LBMM e ELBMM (Patel et al., 2015; Kokilavani et al., 2011), ou utilizá-la em conjunto com outras estratégias, como o algoritmo ILBMM (Rajput e Kushwah, 2016), que usa o min-min em conjunto com algoritmos genéticos, e o Virtual Savant Massobrio et al. (2018), que usa o mesmo para treinar uma inteligência artificial para resolver o problema de escalonamento. Como resultado pretende-se chegar a implementações que sejam tão boas quanto o min-min e também tão eficientes quanto ele. A comparação das heurísticas será realizada usando entradas que considerem três propriedades chave do ambiente: heterogeneidade da máquina, heterogeneidade da tarefa e consistência.

No próximo capítulo, testamos essa implementação híbrida da heurística min-min em um ambiente em nuvem. Para isso, criamos um ambiente simulado de computação heterogênea, utilizando o CloudSim. Neste ambiente a heurística min-min é comparada com a heurística padrão da ferramenta.

CAPÍTULO

4

Simulação do Escalonamento de Tarefas em um Ambiente de Computação em Nuvem

Computação em nuvem é um sistema distribuído e paralelo no qual vários tipos de recursos são dinamicamente alocados aos usuários seguindo o acordo de nível de serviços (SLA) estabelecido entre o provedor do serviço e o consumidor. Em computação em nuvem, software, plataforma e infraestrutura são oferecidos aos consumidores como um serviço (Armbrust et al., 2010).

O uso de simulação é comum em um ambiente de computação em nuvem, pois os serviços nesses ambientes possuem diferentes composições, configurações e requisitos de implantação, o que torna extremamente desafiador quantificar o desempenho em um ambiente real. Por isso, é comum o uso de simuladores como o CloudSim para realizar essa tarefa (Calheiros et al., 2011).

Com o objetivo de modelar e simular *datacenters* de computação em nuvem com GPUs, **?** projetaram e implementaram uma extensão do simulador CloudSim. Os autores seguiram a mesma arquitetura do CloudSim e a extensão proposta pode ser importada no CloudSim sem nenhuma modificação. Dessa forma, pesquisadores podem estender seus projetos de acordo com sua necessidade.

Neste capítulo, realizamos testes em ambientes simulados de computação em nuvem, utilizando o Cloudsim, e comparamos o algoritmo de escalonamento nativo da ferramenta com as heurísticas min-min e max-min. Com isso, mostramos a importância da utilização de um algoritmo de escalona-

mento eficiente e também que se o *framework* apresentasse outras heurísticas implementadas nativamente, poderia trazer benefícios aos usuários que desejassem melhores escalonamentos. Além disso, realizamos testes de escalonamento de *kernels* em ambientes em nuvem com múltiplas GPUs a fim de identificar possíveis otimizações no escalonamento desses ambientes.

4.1 Ambiente em Nuvem

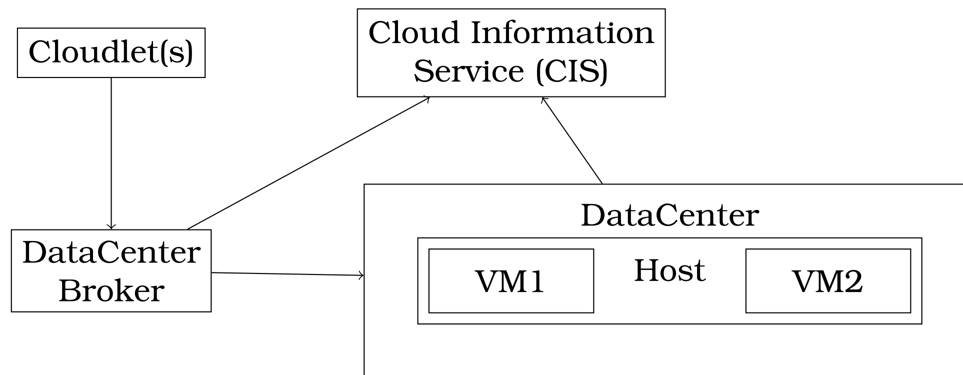
Computação em nuvem é uma abordagem que entrega serviços computacionais de forma segura, tolerante a falhas, sustentável e escalável. Porém para efeito de testes de desempenho, são necessárias metodologias repetitivas e controláveis de avaliação, pois não temos controle sobre todos esses parâmetros. Isso é um dos motivos para utilizar simuladores como o CloudSim para realizar essas tarefas (Calheiros et al., 2011, 2009).

Abaixo são apresentadas as entidades principais de um ambiente em nuvem, de acordo com as nomenclaturas do CloudSim (Calheiros et al., 2009):

1. Task / Cloudlet
2. Cloud Information Service
3. Datacenter Broker
4. Datacenter
5. Host
6. Virtual machine(VM)

O *datacenter broker* é a entidade na qual o escalonamento é implementado. Ele funciona como um intermediário entre o cliente e o provedor do serviço de computação em nuvem. O *datacenter broker* coleta toda informação sobre os recursos disponíveis, a carga nos recursos e os custos de comunicação. Um *datacenter* consiste em vários *hosts*. Cada host pode ter múltiplas máquinas virtuais (VMs). As configurações das VMs são baseadas nas especificações de hardware (poder de processamento, memória, largura de banda, etc) do host em que estão hospedadas. O poder de processamento é calculado em MIPS (Million Instruction Per Second). *Cloud Information Service* (CIS) pode ser definido como um repositório de armazenamento das entidades em nuvem. Uma vez que um *datacenter* é criado, ele tem que ser registrado em um CIS. Quando um usuário requisita um serviço, o *cloudlet* (ou tarefa) é enviado ao *datacenter broker* que coleta as informações dos recursos disponíveis a partir do CIS e então aloca as tarefas a cada máquina virtual. A Figura 4.1 mostra a representação dessas entidades e as relações entre elas.

Figura 4.1: Relação entre as entidades de um ambiente de Computação em Nuvem.



4.1.1 CloudSim

O CloudSim é um *framework* que permite a modelagem, simulação e experimentação de uma infraestrutura de computação em nuvem. Por meio dele podemos criar e interligar as entidades de um ambiente de computação em nuvem, sem custo, com pouco trabalho e conforme desejarmos. Ou seja, podemos simular os diferentes modelos de computação em nuvem: Infraestrutura como um Serviço (*IaaS - Infrastructure as a Service*), Plataforma como um Serviço (*PaaS - Platform as a Service*), e Software como um Serviço (*SaaS - Software as a Service*) (Agarwal et al., 2014).

Utilizando o simulador, pesquisadores e desenvolvedores podem testar o desempenho de novos serviços de aplicações em um ambiente controlado e fácil de configurar. Baseando-se nos resultados reportados pelo CloudSim, eles podem ajustar melhor o desempenho do serviço (Calheiros et al., 2011, 2009). A principal vantagem de usar o CloudSim para os testes de desempenho inicial incluem:

1. **Eficácia do tempo**: isto requer pouco esforço e tempo para implementar um ambiente de teste de aplicação baseado em nuvem.

2. **Flexibilidade e aplicabilidade**: desenvolvedores podem modelar e testar o desempenho dos serviços de aplicações em ambientes em nuvem heterogêneos com pouca programação e tempo de implantação.

O CloudSim é um *framework* escrito em Java, que implementa as entidades do ambiente de computação em nuvem para que o usuário possa instanciá-lo da forma que necessitar. Portanto, por meio desse *framework* é possível configurar uma máquina com configurações semelhantes às do ambiente real que será implementado. É possível, por exemplo, definir o número de instruções por segundo que um *host* é capaz de realizar. Além desse mesmo parâmetro para as máquinas virtuais desse *host*, que são os dispositivos que irão de fato receber e processar as tarefas.

Em geral, as tarefas de diferentes usuários são relativamente independentes: consideramos que há u usuários; t tarefas independentes; m máquinas virtuais; e p *datacenters*. Tarefas podem ser originadas de vários usuários diferentes. Essas tarefas são atribuídas às VMs que vão executá-las. Cada VM é alocada a um *host* em um *datacenter*. Os *datacenters* podem ser compostos por vários *hosts*.

A Figura 4.2 apresenta as entidades envolvidas no CloudSim. Nela, podemos identificar algumas entidades importantes:

- *CIS*. O CIS (*Cloud Information Service*) é a classe que armazena as informações dos estados dos recursos. Quando um usuário requisita um serviço, o *cloudlet* (ou tarefa) é enviado ao *datacenter broker* que coleta as informações dos recursos disponíveis a partir do CIS, e então aloca as tarefas a cada máquina virtual (Calheiros et al., 2011, 2009).

- *DatacenterBroker*. Essa classe modela um broker (intermediário), que é responsável por mediar usuários e provedores de serviço. Algoritmos de escalonamento desenvolvidos pelos usuários são implementados no DatacenterBroker. Logo, os pesquisadores e desenvolvedores de sistema devem estender essa classe (Calheiros et al., 2011, 2009).

- *VmScheduler*. Esta é uma classe abstrata implementada pelo Host. Representa a política de compartilhamento do poder de processamento de um host entre as VMs sendo executadas nele. O funcionamento dessa classe pode ser sobrescrito para se adequar a uma política específica de compartilhamento do processador (Calheiros et al., 2011, 2009).

- *VmAllocationPolicy*. Essa classe representa a política de alocação de VMs aos *hosts*. A política padrão de alocação é selecionar *hosts* disponíveis no *datacenter*, que satisfazem os requisitos de memória, armazenamento e disponibilidade da VM (Calheiros et al., 2011, 2009).

4.1.2 GPUCloudsim

O CloudSim apesar de ser capaz de simular um ambiente de computação em nuvem, não inclui GPUs em sua configuração. Pensando nisso, **?** criaram o GPUCloudsim, uma extensão que inclui esse recurso no simulador. A Figura 4.3 ilustra os principais componentes de um servidor equipado com GPUs e as máquinas virtuais que as implementam.

Um servidor incorpora uma ou mais CPUs e zero ou mais GPUs. O *hypervisor* mantém o controle de todas as VMs no servidor e tem a responsabilidade de virtualizar os recursos do servidor. Isto inclui prover e escalonar recursos.

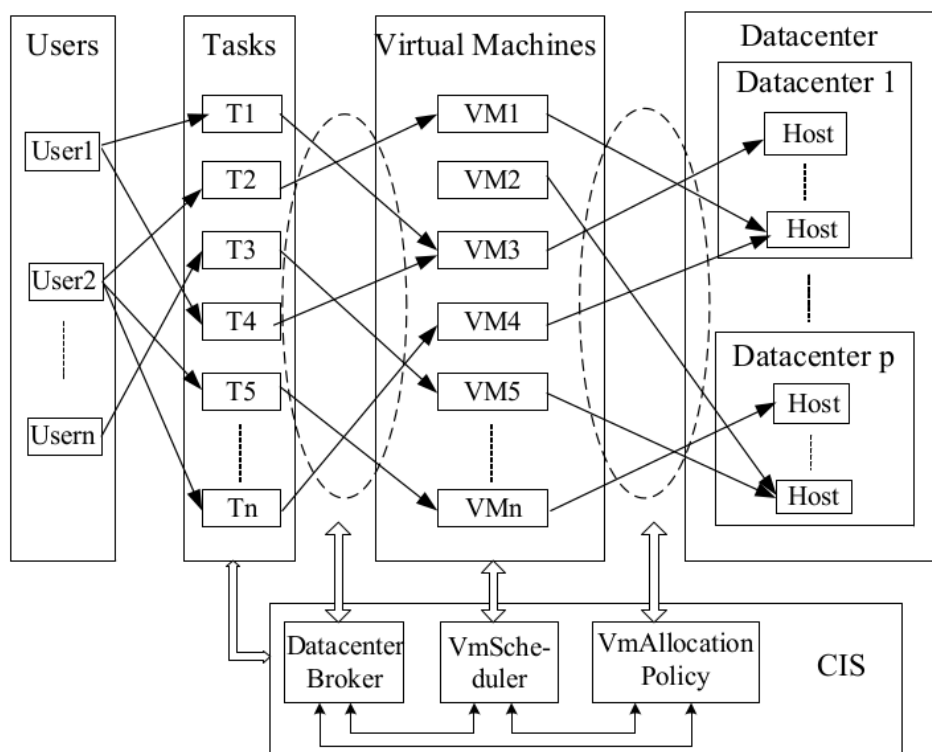


Figura 4.2: Funcionamento do CloudSim (Li et al., 2011).

Uma VM requer uma ou mais CPUs e uma ou nenhuma GPU. Além disso, VMs contém as aplicações do usuário com tarefas que rodam na CPU e na GPU (**?**). Agora, quando uma máquina virtual é criada, ela pode utilizar recursos de CPU e RAM, como o CloudSim, mas também recursos de GPU.

As GPUs são alocadas nos dispositivos *multicore*, que podem ter várias GPUs. Os dispositivos *multicore* são conectadas no sistema usando barramento *full-duplex* PCI Express de alta velocidade (PCIe), que é compartilhado entre as GPUs do dispositivo. Mecanismos de cópia são responsáveis por transferir dados entre o *host* e o *device*, isto é, CPU e GPU. Um mecanismo de cópia move dados em uma direção por vez. Com dois mecanismos, uma GPU é capaz de mover dados simultaneamente na duas direções.

A Figura 4.4 (a) mostra os componentes de uma placa de vídeo dual. Um *switch* é necessário para conectar as GPUs ao barramento PCIe. Cada GPU tem sua própria memória física global e consiste em vários núcleos SIMD (Single Instruction Multiple Data) chamados Streaming Multiprocessor (SMs), que são os responsáveis por executar os blocos de *threads* das tarefas em execução. A *GigaThread engine* é responsável por escalonar os blocos de *threads* nos SMs. Os blocos de *threads* são distribuídos igualmente nos SMs e permanecem lá até que terminem sua execução. Isso sugere que os SMs não podem ser compartilhados entre os blocos de *threads*. As placas gráficas da NVIDIA disponibilizam até 8 blocos em execução por SM e com tecnologia Hyper-Q permitem que os *kernels* compartilhem recursos dos SMs por meio de filas

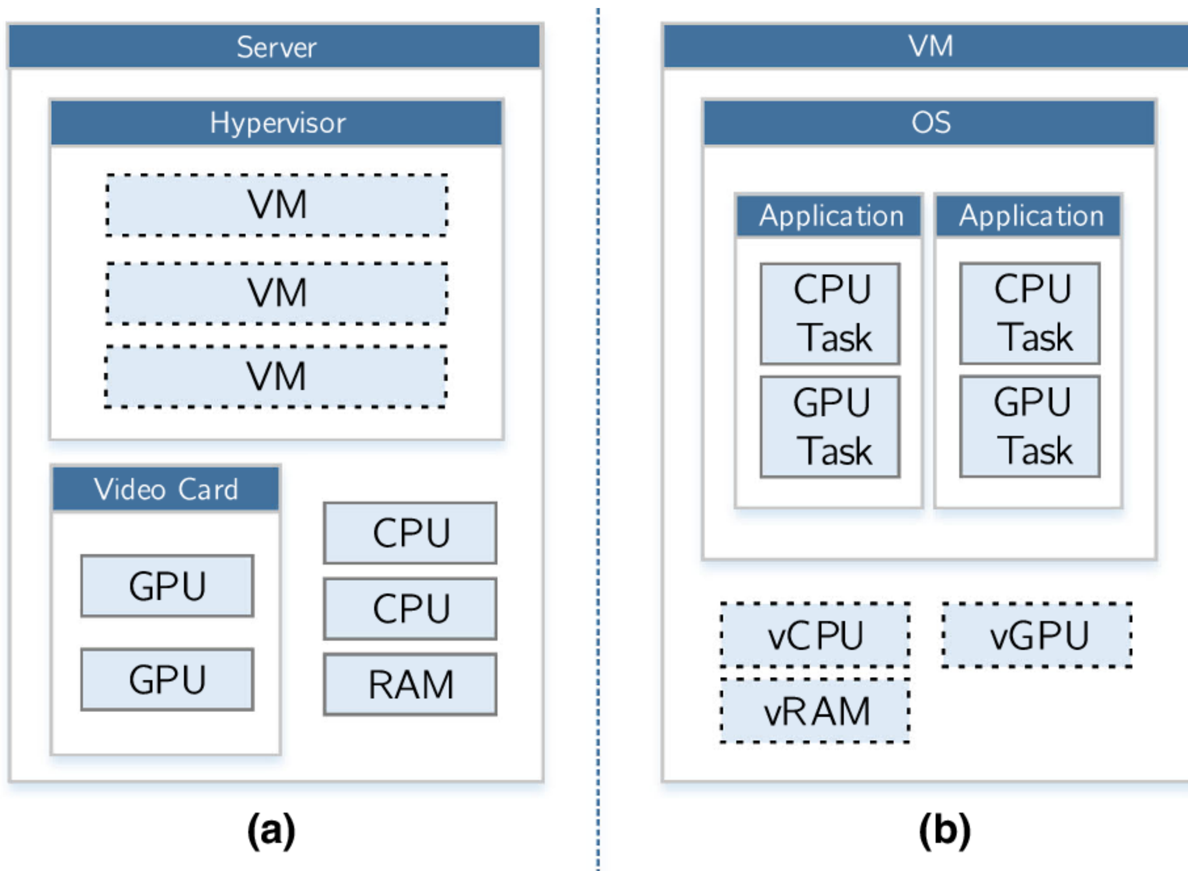


Figura 4.3: (a) Principais componentes de um servidor equipado com GPU e (b) as VMs que suportam GPUs (**?**).

de execução (32 *streams*), porém esse recurso não foi implementado no GPU-Cloudsim.

Para executar uma tarefa na GPU, os dados de entrada devem ser transferidos para o dispositivo. Depois, a tarefa é executada por um grande número de *threads* particionadas em blocos. Finalmente, os dados de saída são transferidos de volta para o *host*, conforme mostrado na Figura4.4.

Abaixo são apresentadas as principais classes do GPUCloudsim para o desenvolvimento desse trabalho:

- **GpuVm**. Estende a classe VM do CloudSim para modelar uma VM com vGPU.
- **Vgpu**. Representa a GPU virtual que foi alocada na VM.
- **GpuHost**. Estende a classe *Host* para modelar um *Host* com uma ou mais placas de vídeo.
- **VgpuScheduler**. Define a política de compartilhamento do poder de processamento da placa de vídeo entre suas vGPUs.
- **Pgpu**. Representa uma GPU física. A Pgpu incorpora múltiplos elementos de processamento (Pe) que representam os SMs da GPU.

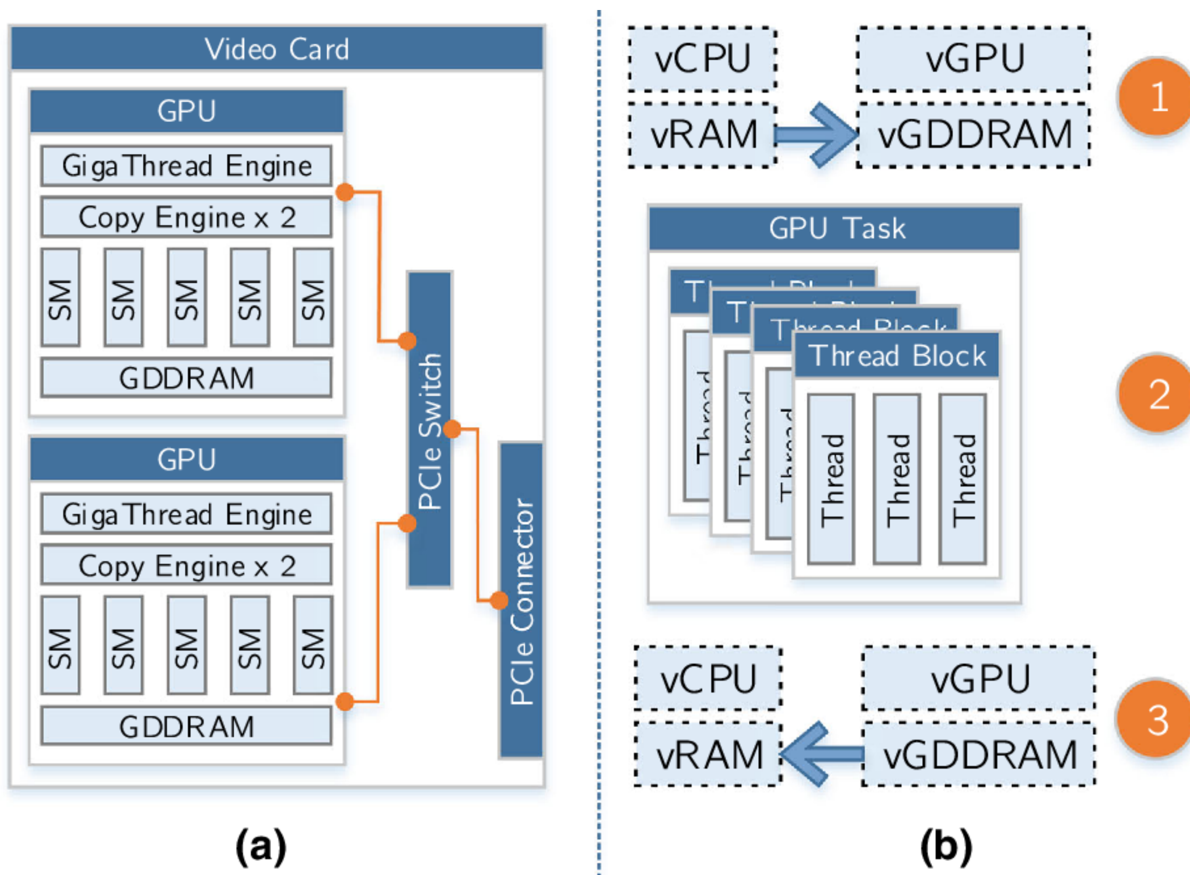


Figura 4.4: (a) Placa de vídeo e os recursos das GPUs. (b) Os passos de execução de uma tarefa na GPU (**?**).

- **VideoCard**. Representa um dispositivo *multicore* (placa de vídeo) do sistema. Um dispositivo *multicore* tem um ou mais Pgpu no dispositivo, cada um com sua própria memória.

- **Pe**. Representa um elemento de processamento. Simula um SM na GPU.

- **PeProvisioner**. Representa a política de provimento que é usado para alocar o poder de processamento de um Pe às vGPUs.

- **GpuCloudlet**. Estende a classe Cloudlet do CloudSim para representar uma aplicação com porções de execução na CPU e na GPU.

- **GpuTask**. Representa a computação que é realizada na GPU. Isso inclui o número de blocos de *threads* e o tamanho de cada bloco. Incorpora o modelo de utilização para imitar as demandas por recursos.

- **GpuTaskScheduler**. Representa a política de escalonamento adotada por uma vGPU para executar suas tarefas na GPU.

4.2 Simulação do Escalonamento de Tarefas usando o Cloudsim

Em 2001 Braun et al. (2001) fizeram uma comparação entre onze heurísticas diferentes. Em seus testes, utilizaram cenários com características de heterogeneidade consistente e inconsistente, bem como heterogeneidade de máquina alta e baixa. Os resultados mostraram que essas características afetam diretamente a qualidade da solução gerada por essas heurísticas. Além disso, pode-se evidenciar também que os algoritmos genéticos (GA) tiveram os melhores resultados na maioria dos cenários, mas que o min-min pode ser resolvido muito rapidamente com resultados próximos ao GA.

Etminani e Naghibzadeh (2007) desenvolveram um algoritmo que seleciona a melhor solução entre as heurísticas min-min e max-min em cada iteração do algoritmo, baseado na utilização das máquinas e no tamanho das tarefas que ainda precisam ser escalonadas. Os resultados mostraram que essa heurística é melhor que as heurística min-min e max-min, quando aplicadas isoladamente. O novo algoritmo reduziu o *makespan*, aumentou a utilização das máquinas e o balanceamento de carga.

Uma adaptação da heurística min-min foi proposta por Kokilavani et al. (2011) para melhorar o balanceamento de carga. Após aplicar o min-min, o algoritmo percorre as máquinas analisando a carga de cada uma e fazendo o reescalonamento quando necessário. Para isso, encontra a máquina com o maior *makespan* e a tarefa dentro dela que possui o menor tempo esperado. Depois procura qual máquina irá gerar o maior tempo de conclusão para essa tarefa. Se o maior tempo de conclusão for menor que o *makespan*, reescalona essa tarefa na nova máquina e calcula o novo tempo. Esses passos são repetidos até que todas as máquinas tenham sido consideradas.

Kumar e Verma (2012), fizeram a comparação entre um algoritmo genético padrão, que utiliza uma população inicial gerada aleatoriamente e um algoritmo genético modificado, que utiliza como população inicial as heurísticas max-min e min-min. Eles utilizaram o CloudSim para validar o algoritmo e concluíram que um algoritmo genético consegue resultados melhores quando a sua população inicial já é uma boa solução para o problema sendo resolvido.

Bhoi e Ramanuj (2013) fizeram uma alteração no algoritmo *Improved Max-min* de Elzeki et al. (2012) para no lugar de selecionar as tarefas maiores para serem escalonadas primeiro, escolhe as tarefas que possuem a média do tempo de execução das tarefas que faltam escalonar ou que estão um pouco acima da média. A tarefa selecionada pelos algoritmos em cada passo é atribuída à máquina que possui o menor tempo de conclusão. A esse algoritmo deram o nome de *Enhanced Max-min* e conseguiram mostrar, por meio de tes-

tes no simulador CloudSim, que essa variação no algoritmo proporciona um *makespan* melhor.

Posteriormente, Tsai et al. (2014) criaram o Hyper Heuristic Scheduling Algorithm (HHSA). Um algoritmo que seleciona aleatoriamente uma heurística a partir do conjunto de heurísticas disponíveis, e executa o número de iterações que satisfaz seus critérios. A cada execução são verificados se critérios de modificação e melhorias estão sendo satisfeitos para aquela heurística. Caso contrário, seleciona aleatoriamente uma nova. Os resultados mostraram que esse método consegue convergir mais rapidamente que os outros testados e apresentou os melhores resultados para a maioria dos casos.

4.2.1 Experimentos e Estratégias de Escalonamento

O problema de escalonamento de tarefas em ambientes de computação heterogênea é um problema NP-difícil, por isso um grande número de heurísticas foram propostas para obter soluções semi-ótimas para ele. Nesta seção, são apresentadas as heurísticas min-min e max-min, que serão abordadas nesse trabalho, e que são comumente utilizadas na solução desse problema. Além disso, pode-se evidenciar também que os algoritmos genéticos (GA) tiveram os melhores resultados na maioria dos cenários, mas que o min-min pode ser resolvido muito rapidamente com resultados próximos ao GA (Braun et al., 2001).

Min-min. A heurística min-min começa com o conjunto U de todas as tarefas não mapeadas. Então, o conjunto de menor tempo de conclusão M para cada tarefa em U é encontrado analisando os tempos das tarefas já atribuídas às máquinas até aquele instante e o tempo das tarefas em U. A tarefa cujo tempo de conclusão seja o menor é escolhida e atribuída à máquina correspondente. A nova tarefa mapeada é removida do conjunto U e o processo se repete até que todas as tarefas sejam mapeadas (Braun et al., 2001).

Max-min. A heurística max-min também começa com o conjunto U de tarefas não mapeadas, encontra o conjunto de menor tempo de conclusão M para cada tarefa em U, analisando os tempos das tarefas já atribuídas às máquinas até aquele instante e o tempo das tarefas em U. Posteriormente, a tarefa cujo tempo de conclusão seja máximo é selecionada e atribuída à máquina que proporciona o menor tempo de conclusão. A nova tarefa mapeada é removida do conjunto U e o processo se repete até que todas as tarefas sejam mapeadas (Kumar e Verma, 2012).

Para testar as estratégias de escalonamento no CloudSim abordamos três características:

1. características das tarefas,

2. características das máquinas e

3. características do algoritmo de escalonamento.

Para avaliar essas características usamos 128, 256, 512, 1024 e 2048 tarefas, geramos 10 arquivos para cada uma dessas dimensões. Esses arquivos são compostos pelo números de instruções de cada tarefa em cada linha do arquivo. O número de instruções de cada tarefa foi gerado de forma aleatória no intervalo de 100 até 10000 instruções. A Tabela 4.1 mostra a média e o desvio padrão do número de instruções gerados para cada uma das 10 entradas para as dimensões 1024 e 2048.

	1024 Tarefas		2048 Tarefas	
	Média	Desvio Padrão	Média	Desvio Padrão
Entrada 1	5004,7227	2825,8179	5153,4551	2927,0715
Entrada 2	4999,2402	2859,6238	5259,6758	2831,0745
Entrada 3	5085,7295	2918,1625	4942,0654	2837,4149
Entrada 4	5089,9541	2857,8014	4973,4033	2796,6533
Entrada 5	4908,5400	2848,9555	5033,8291	2867,6947
Entrada 6	5004,3115	2888,7416	5132,0957	2860,0561
Entrada 7	5024,6641	2881,7100	4817,8721	2794,7192
Entrada 8	5072,2148	2848,7863	4965,8984	2854,9785
Entrada 9	4908,9268	2741,1339	4908,6230	2876,5185
Entrada 10	4975,2432	2888,5286	5133,0479	2902,4015

Tabela 4.1: Média e o Desvio Padrão do Número de Instruções Gerados nos Arquivos de Entrada.

Para avaliar as características das máquinas, utilizamos 2 cenários. No primeiro, a quantidade de máquinas aumenta junto com a quantidade de tarefas, dessa forma, a relação tarefas-máquinas é sempre a mesma em todos os testes realizados. No segundo cenário a quantidade de tarefas é fixa e a quantidade de máquinas varia, porém, tomamos o cuidado de manter o mesmo poder computacional em todos os testes, ou seja, à medida que o número de máquinas aumenta, o poder computacional de cada uma diminui. Em ambos os cenários, os *hosts* recebem VMs de acordo com sua capacidade computacional em MIPS, ou seja, enquanto o somatório dos MIPS das VMs alocadas em um *host* não superar o MIPS do host, ele recebe mais VMs.

Utilizamos a heurística min-min, que procura a cada iteração o par tarefa/máquina que gera o menor *makespan*, a heurística max-min, que procura a tarefa com maior *makespan* e escalona na máquina que a executa mais rápido, e o algoritmo padrão de escalonamento do Cloudsim, que distribui as tarefas igualmente entre as máquinas.

Cenário 1: Relação entre Tarefas e Máquinas Fixa

Nesse cenário, o número de tarefas aumenta na mesma proporção que a quantidade de máquinas. Por exemplo, no teste com 4 máquinas virtuais, existem 128 tarefas a serem escalonadas, já quando temos 8 máquinas virtuais, são 256 tarefas a serem escalonadas. Ao dividir o número de tarefas pelo número de máquinas a proporção é sempre 32 tarefas para cada máquina em todos os casos de testes.

Criamos 6 *datacenters*, cada um com 6 *hosts* e poder computacional igual a 1000 MIPS, e as VMs foram alocadas conforme cabiam nos *hosts*. Dessa forma, pode haver mais de uma VM alocada em um mesmo *host* e também *hosts* sem VMs alocadas. Fizemos a análise a partir de 4 máquinas virtuais com poder computacional (em MIPS) de: 1000, 500, 250 e 125. De um teste para o outro o número de máquinas virtuais com cada uma dessas velocidades duplica, conforme descrito na Tabela 4.2.

Tabela 4.2: Cenário 1: Relação entre Tarefas/Máquinas igual em todos os testes.

Quantidade de Máquinas Virtuais						
MIPS 1000	MIPS 500	MIPS 250	MIPS 125	Total de Máquinas Virtuais	Total de Tarefas	Total de Datacenter
1	1	1	1	4	128	1
2	2	2	2	8	256	1
4	4	4	4	16	512	2
8	8	8	8	32	1024	3
16	16	16	16	64	2048	6

Cenário 2: Poder Computacional do Ambiente e Quantidade de Tarefas Fixos

Nesse cenário, o somatório dos MIPS das máquinas virtuais em todos os testes é sempre igual a 2000, independente do número de máquinas virtuais. Logo, conforme o número de máquinas virtuais aumenta, o MIPS de cada uma diminui.

A Tabela 4.3 descreve o poder computacional em MIPS de cada VM e dos *hosts* de cada *datacenter*. Criamos 5 *datacenters*, cada um com 3 *hosts* com poder computacional igual ao da VM mais rápida. Por exemplo, os *hosts* do teste com 4 VMs, foram criados com MIPS igual a 550 e as VMs foram alocadas conforme cabiam em cada *host*, dessa forma, pode haver mais de uma VM alocada em um mesmo *host* e também *hosts* sem VMs alocadas.

Realizamos os experimentos em uma máquina com processador Intel(R) Xeon(R) CPU E5-1620 v3 3.50GHz, 32 GB de RAM, placa de processamento gráfico Quadro M4000 com 8 GB de memória global, versão do CUDA 8.0 e

Tabela 4.3: Cenário 2: Poder computacional do ambiente e quantidade de tarefas iguais em todos os testes.

Quantidade de Datacenter	Quantidade de VMs	MIPS de cada VM	MIPS de cada Host	Total de Tarefas
1	2	1100, 900	1100	1024
2	4	550, 525, 475, 450	550	1024
3	8	265, 262, 257, 253, 248, 244, 239, 232	265	1024
5	16	160, 158, 153, 148, 144, 137, 132, 129, 126, 117, 113, 107, 101, 94, 93, 88	160	1024

versão 5.4 do compilador gcc. Os códigos dos programas e arquivos de entrada estão disponíveis no GitHub[1].

4.2.2 Resultados e Análises

Nesta seção apresentamos os resultados obtidos e a análise do comportamento das heurísticas em cada um dos experimentos.

Cenário 1: Relação entre Tarefas e Máquinas Fixa

A Tabela 4.4 apresenta a média e o desvio padrão do *makespan* das 10 entradas em cada dimensão do problema. O desvio padrão das heurísticas min-min e max-min vai diminuindo à medida que a dimensão do problema aumenta, o que indica que a diferença entre as soluções vai diminuindo. Isso já não ocorre com o algoritmo padrão de escalonamento do Cloudsim, pois ele não leva em conta o tempo de processamento das tarefas para escalonar.

Tabela 4.4: Média e desvio padrão dos *makespans* para cada dimensão do problema.

	Min-min		Max-min		Padrão do Cloudsim	
Dimensão	*Makespan*	Desvio Padrão	*Makespan*	Desvio Padrão	*Makespan*	Desvio Padrão
4x128	326,39	28,70	329,41	22,19	1249,04	212,65
8x256	351,59	11,18	350,22	10,04	1300,80	186,81
16x512	347,03	10,92	344,78	10,07	1320,26	110,53
32x1024	342,91	8,31	342,95	4,39	1246,94	99,68
64x2048	347,79	8,35	347,66	5,99	1296,50	112,72

O *speedup* das heurísticas min-min e max-min quando comparadas com o escalonamento padrão do Cloudsim é apresentado na Tabela 4.5. Ela mostra

[1] https://github.com/rafaelfschmid/cloudsim-scheduling.git

que, ao usar qualquer uma das heurísticas, o desempenho fica pelo menos 3,6 vezes melhor em qualquer dimensão do problema.

Na Tabela 4.6 apresentamos os tempos, em segundos, que as heurísticas max-min e min-min levam para definir o escalonamento das tarefas nas máquinas disponíveis. Como as dimensões do problema não são grandes e estamos usando um algoritmo híbrido GPU/CPU, o *overhead* da chamada da função em GPU toma praticamente todo o tempo do escalonamento, por isso o tempo de escalonamento é quase igual em todas as dimensões.

Tabela 4.5: *Speedup* em relação ao algoritmo padrão do Cloudsim.

	Speedup	
Dimensão	Min-min	Max-min
4x128	3,8268	3,7917
8x256	3,6998	3,7143
16x512	3,8044	3,8293
32x1024	3,6363	3,6360
64x2048	3,7279	3,7293

Tabela 4.6: Tempo de Escalonamento das Heurísticas para as Dimensões do Problema

	Tempo (s)	
Dimensão	Min-min	Max-min
4x128	0,5683	0,5551
8x256	0,5646	0,5543
16x512	0,5695	0,5608
32x1024	0,5912	0,5784
64x2048	0,5913	0,5932

As médias do *makespan* obtidos em cada dimensão, já somado o tempo do escalonamento, são apresentados na Figura 4.5. As heurísticas min-min e max-min possuem *makespan* muito próximos. O ambiente testado é consistente, isto é, se uma máquina m_j executa uma tarefa t_i mais rápido que outra máquina m_k, ela executará qualquer tarefa mais rápida que m_k. Ou seja, no ambiente testado, a máquina mais rápida, será a mais rápida para executar qualquer tarefa. Portanto, a diferença entre os algoritmos min-min e max-min é a ordem de escolha das tarefas. Enquanto o max-min começará o escalonamento pelas tarefas maiores, o min-min começará pelas tarefas menores.

Cenário 2: Poder Computacional e Quantidade de Tarefas Fixos

Um ponto que chama a atenção na Figura 4.5 é que os *makespans* das heurísticas min-min e max-min quase não são afetados quando aumentamos a dimensão do problema. Por isso, foi criado o cenário 2 para verificar o comportamento dessas heurísticas ao se manter o poder computacional e a quantidade de tarefas, enquanto aumentamos a quantidade de máquinas.

Identificamos que, quando se mantém o poder computacional, o tempo para resolver o mesmo problema com mais máquinas, usando min-min ou max-min, são muito próximos. Ao analisar a Tabela 4.7 verifica-se que, para todas as dimensões, o max-min conseguiu finalizar as tarefas com uma média de 2564 segundos, alterando apenas os decimais. Já o min-min, apesar de apresentar diferenças de uma dimensão para outra, elas são muito pequenas,

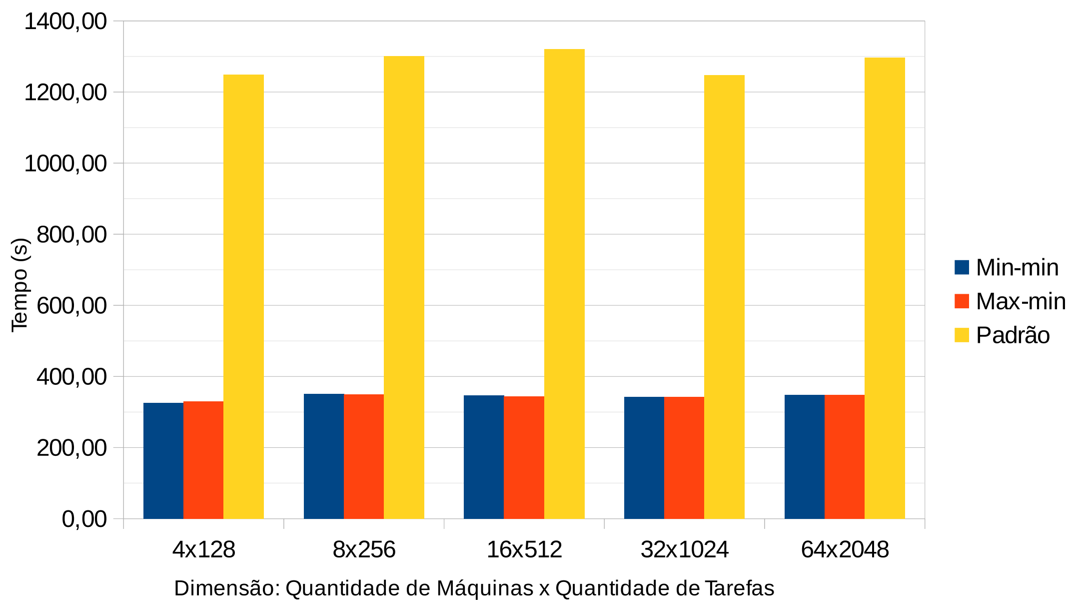


Figura 4.5: Média do Tempo de Execução de Cada Heurística para as Dimensões do Problema.

sendo que o maior tempo foi de $2575,03$ e o menor foi $2559,08$, o que equivale a menos de 1% de diferença.

Tabela 4.7: Média e Desvio Padrão dos *Makespans* para cada Dimensão.

	Min-min		Max-min		Padrão do Cloudsim	
Dimensão	*Makespan*	Desvio Padrão	*Makespan*	Desvio Padrão	*Makespan*	Desvio Padrão
2x1024	2563,75	34,39	2564,94	33,29	2826,18	71,08
4x1024	2565,16	33,52	2564,71	33,19	2806,54	82,55
8x1024	2559,08	32,32	2564,54	33,26	2756,43	119,82
16x1024	2575,03	32,87	2564,54	33,33	3429,60	114,71

Outra análise efetuada foi a comparação da quantidade de tarefas atribuídas a cada VM. Como o algoritmo do Cloudsim atribui a mesma quantidade de tarefas para todas as máquinas, ele não é apresentado nessa análise. Para comparar as heurísticas min-min e max-min, foi criada a Figura 4.6 que apresenta, para uma das entradas com 16 máquinas virtuais e 1024 tarefas, a quantidade de tarefas em cada VM. Como o poder computacional da VM 0 é maior que o poder computacional da VM 1 e assim por diante, a suposição é que VMs com IDs menores recebam mais tarefas que VMs com IDs maiores, e que foi exatamente o que aconteceu em ambas as heurísticas. A diferença entre o min-min e o max-min é que a quantidade de tarefas das máquinas mais lentas é um pouco maior no max-min do que no min-min, e, consequentemente, o número de tarefas das máquinas mais rápidas é um pouco menor. Isso ocorre porque o max-min é melhor para fazer o balanceamento de carga, já que resolve as tarefas maiores primeiro.

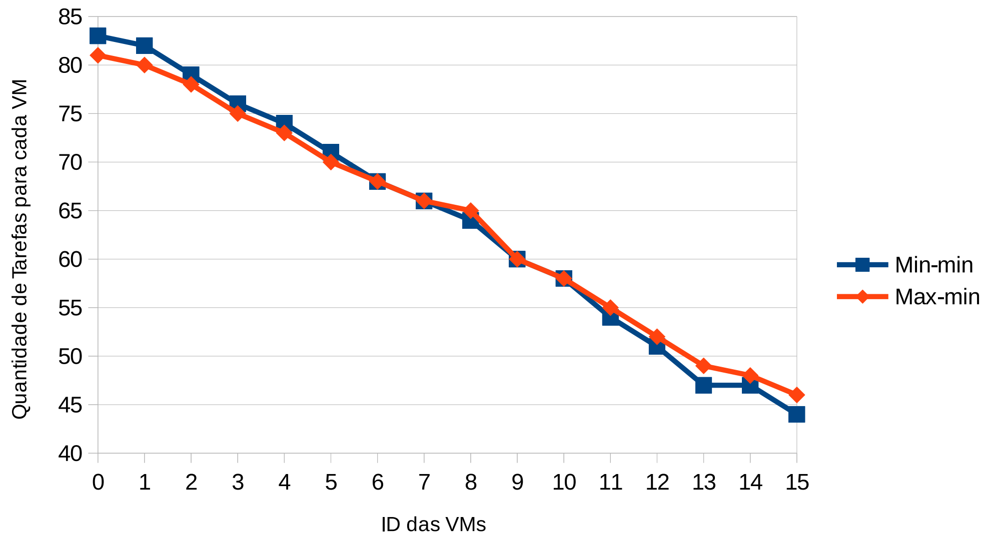


Figura 4.6: Quantidade de Tarefas em Cada VM.

Apesar de as máquinas mais rápidas receberem mais tarefas, os métodos min-min e max-min distribuem as tarefas de acordo com o *makespan* de cada máquina, ou seja, eles atribuem a tarefa sendo escalonada à máquina que vai terminá-la primeiro, considerando-se as tarefas que já foram atribuídas a ela. A Figura 4.7 apresenta o *makespan* de cada VM utilizando uma das entradas com 16 máquinas virtuais e 1024 tarefas. Nela é possível perceber que o algoritmo max-min efetua melhor o balanceamento de carga, por começar pelas tarefas maiores, aquelas que resultarão em maior desbalanceamento se deixadas para o final, que é o que ocorre com a heurística min-min. O algoritmo padrão do Cloudsim distribui o mesmo número de tarefas para cada máquina, independente de seu tamanho, logo, é esperado que as máquinas mais lentas (com ID maiores) levem mais tempo para finalizarem seus trabalhos.

A Tabela 4.8 apresenta o percentual de utilização dos Datacenters. Para a obtenção dessa medida efetuamos a soma do MIPS de todas as VMs alocadas em um *Datacenter* e dividimos pelo total de MIPS dos *hosts* existentes nele. Por exemplo, no caso de teste com 8 VMs, foram alocados 3 *datacenters*, cada um com 3 *hosts* com poder computacional de 265 MIPS. No primeiro *datacenter* foram alocadas 3 VMs, com MIPS iguais a: 265, 262 e 257. Sendo assim, a utilização desse *datacenter* foi o somatório dos MIPS das 3 VMs dividido pelo somatório dos MIPS dos 3 *hosts*:

$$\frac{265+262+257}{265+265+265} = 0,9862 = 98,62\%.$$

Essa tabela se aplica a todos os algoritmos, pois o algoritmo de alocação de VMs a um Host utilizado foi o padrão do Cloudsim, que aloca em ordem conforme as VMs vão cabendo em um *Host*.

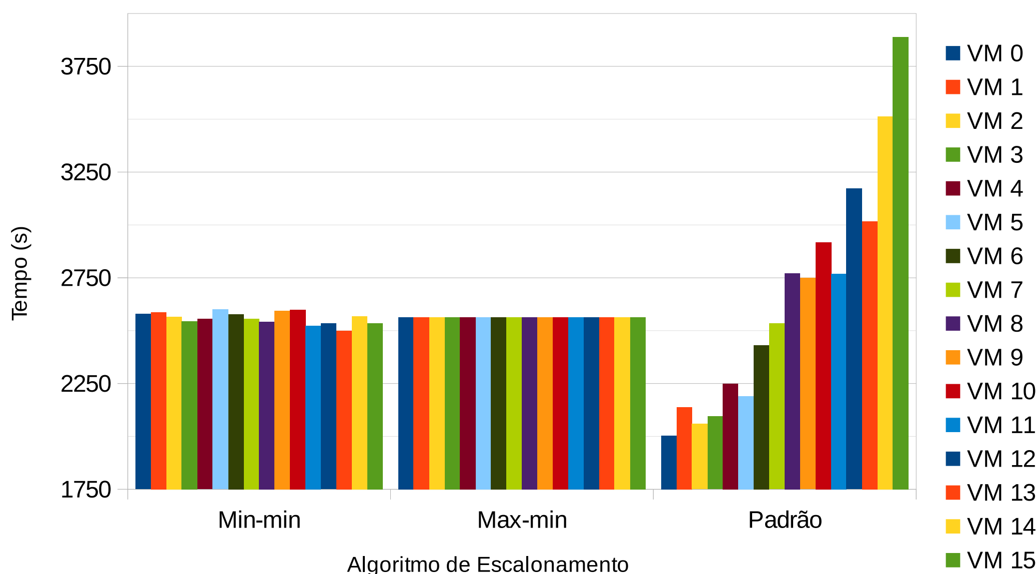


Figura 4.7: Tempo de Execução de Cada VM para uma das Entradas de Dimensão 16 máquinas e 1024 tarefas.

Na Tabela 4.8 também é apresentado um cenário hipotético no qual o cálculo do preço de um *datacenter* leva-se em conta o percentual de utilização do seu potencial de processamento, ou seja, se tiverem MIPS não utilizados, eles não são cobrados no preço final. Para simplificar as análises, consideramos o custo de 1 *datacenter* como sendo 100. Dessa forma, para que todos os ambientes custem o mesmo, os preços de cada *datacenter* devem seguir os preços sugeridos na tabela. Esses valores foram obtidos dividindo a utilização de 1 datacenter pelo somatório das utilizações dos outros *datacenters*. Por exemplo, para calcular o custo de cada *datacenter* ao utilizarmos 5 *datacenters* com baixo poder computacional, dividimos 60,61 (que é a utilização de 1 *datacenter* sozinho) por 416,67 (que é o somatório das utilizações dos *datacenters*): $60,61/416,67 = 0,1455$ (15,55 %).

Tabela 4.8: Percentual de Utilização dos Datacenters e Preço Hipotético Sugerido

	Número de Datacenters			
	1	2	3	5
Datacenter 0	60,61%	93,94%	98,62%	98,13%
Datacenter 1		27,27%	93,71%	89,38%
Datacenter 2			59,25%	80,63%
Datacenter 3				88,54%
Datacenter 4				60,00%
Utilização Total	60,61%	121,21%	251,57%	416,67%
Preço Sugerido de cada Datacenter	100,00	50,00	24,09	14,55

As implementações apresentadas nesta seção, fazem a distribuição de tare-

fas em diversas máquinas. Quando utilizamos GPUs, não só a distribuição das tarefas entre as máquinas é importante, mas também a ordem em que elas serão executadas em cada GPU. Uma função que executa na GPU é chamada de Kernel, e as GPUs mais modernas suportam vários *kernels* sendo executados ao mesmo tempo, desde que haja recursos de hardware disponíveis. Dessa forma, na próxima seção é introduzido outro problema de escalonamento, o problema da reordenação de *kernels* em GPU.

4.3 Simulação do Escalonamento de Kernels em GPUs usando o GPUCloudsim

Nesta seção, apresentamos os resultados do escalonamento de *kernel* em um ambiente simulado de computação em nuvem, usando o GPUCloudSim. Avaliamos o escalonamento padrão da ferramenta e um algoritmo utilizando mochila unidimensional.

O GPUCloudSim não implementa o compartilhamento de SMs entre *kernels* diferentes, mas implementa o compartilhamento de GPU entre os *kernels*, permitindo execução paralela entre dois *kernels* quando existem SMs livres. Para superar essa limitação, utilizamos o número de SMs da GPU para definir a quantidade de recursos disponíveis e o número de blocos dos *kernels* indicando a quantidade utilizada desse recurso por cada um deles. Para isso, o número máximo de blocos de um *kernel* e o número de SMs da GPU foram limitados em 32. Dessa forma, quando um *kernel* tiver 32 blocos, ele vai usar todos os recursos da GPU e não poderá ser compartilhado com outro kernel. Porém, quando esse número for menor que 32, será possível executar paralelamente outros *kernels*. Isto é, o número de blocos sendo executados dividido pelo número de SMs define o percentual de utilização dos recursos da GPU pelo kernel.

Ao estudar a extensão do simulador, identificamos que os blocos de *threads* eram distribuídos entre os SMs da GPU conforme disponibilidade. Porém, ao atribuir uma quantidade de SMs a um kernel, essa quantidade se mantinha até o final da execução daquele kernel. Porém, quando um *kernel* terminava, liberando assim novos SMs da GPU, *kernels* que já estavam em execução não podiam utilizar esses SMs livres. Dessa forma, realizamos um ajuste na classe GpuTaskSchedulerLeftover, para suprir essa necessidade.

Nas próximas seções, apresentamos os resultados da simulação de *kernels* executando em três ambientes diferentes. O primeiro, contendo apenas uma GPU, em que os *kernels* deveriam ser reordenados, ou seja, o objetivo era otimizar a utilização da GPU. O segundo, contendo várias máquinas com GPUs, com as mesmas configurações de hardware. E a terceiro ambiente, contendo

várias máquinas com GPUs, em que as configurações de hardware eram diferentes.

4.3.1 Ambiente com uma única GPU

Os testes simulados com uma única GPU foram realizados com dois algoritmos. O primeiro, chamado algoritmo padrão, que executa os *kernels* na ordem em que chegam para execução, e o algoritmo da mochila, que considera os recursos computacionais para reordenação de *kernels*.

Algoritmos

O algoritmo padrão, simplesmente executa os *kernels* na ordem em que aparecem, e o paralelismo acontece sempre que tiverem SMs livres para o próximo *kernel* que está chegando. Por exemplo, em uma GPU com 32 SMs que precisa executar três *kernels* contendo cada um: 22, 10 e 3 blocos, seria possível escalonar os dois primeiros *kernels* juntos, pois totalizam 32 blocos, e o terceiro iniciaria assim que o primeiro deles terminasse, pois haveriam SMs livres.

Já o algoritmo da mochila unidimensional utiliza o número de SMs da GPU como o peso máximo suportado pela mochila e o número de blocos dos *kernels* como o peso de cada item a ser colocado na mochila. Um problema da mochila unidimensional é resolvido em cada iteração do algoritmo. Na primeira execução o peso máximo suportado pela mochila é o número total de SMs da GPU, e nas próximas iterações, conforme *kernels* vão finalizando e liberando SMs, um novo problema da mochila é resolvido considerando apenas esses recursos disponíveis.

Resultados

Para simular o teste no GPUCloudsim, criamos um ambiente de computação em nuvem com um *datacenter*, um *host*, uma máquina virtual (VM) e uma GPU. Os tempos de execução gerados pelos algoritmos estão descritos na Tabela 4.9. Nas colunas estão representados as quantidades de *kernels* sendo escalonados, e nas linhas estão descritos os tempos de execução gerados por cada algoritmo.

Tabela 4.9: Tempos de execução em uma GPU.

	Quantidade de Kernels					
Algoritmo	8	16	32	64	128	256
P	13,6	26,81	53,43	107,55	223,79	478,9
MT	12,22	20,18	38,04	79,14	159,16	384,54

A implementação da mochila (**MT**) foi melhor em todos os cenários testados, apresentando uma melhora de pelo menos 11% em relação à implementação padrão (**P**), e uma melhora média de 30%. A Figura 4.8 apresenta o *speedup* da versão da mochila (**MT**) quando comparada com a versão padrão (**P**). Para uma quantidade pequena de *kernels*, o *speedup* acaba sendo pequeno, porém entre 16 e 128 *kernels* a melhora é de aproximadamente 40%, descrescendo para 20% nos testes com 256 *kernels*. Isto ocorre, porque a estimativa de tempo de execução de cada *kernel* não é precisa, então à medida que o número de *kernels* aumenta, o erro também aumenta.

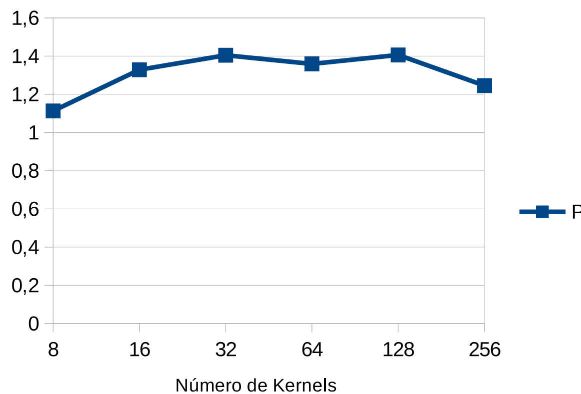


Figura 4.8: *Speedup* da implementação **MT** em relação ao escalonamento padrão **P**.

4.3.2 Ambiente com Múltiplas GPUs

Um ambiente de computação em nuvem pode ser composto de diversas máquinas trabalhando em conjunto para a solução de um problema específico, da mesma forma, um ambiente pode ser composto por múltiplas GPUs trabalhando em conjunto para a resolução de um problema. Para explorar esse problema, criamos um ambiente simulado em nuvem, usando o GPU-CloudSim, com diversas GPUs e vários *kernels* a serem escalonados.

No ambiente com múltiplas GPUs, escalonamos um número de *kernels* proporcional ao número de máquinas, para que fosse coerente a comparação com o ambiente com uma única GPU. Isto é, quando aumentamos o número de GPUs, o número de *kernels* aumenta na mesma proporção. Por exemplo, escalonar 8 *kernels* no ambiente com uma única GPU equivale a escalonar 16 *kernels* no ambiente com 2 GPUs, distribuindo igualmente esses *kernels*, resultaria em 8 *kernels* por GPU. Neste ambiente implementamos quatro algoritmos: padrão (**P**), mochila (**MT**), heurística min-min (**Min**) e a união do min-min com a mochila (**MinMT**).

Algoritmos

O algoritmo padrão (**P**), distribui os *kernels* entre as máquinas na ordem em que aparecem e executa os mesmos também na ordem em que aparecem. Dessa forma, além de não aproveitar o paralelismo entre os *kernels* em cada uma das máquinas, pois não explora as possibilidades de execução paralela, subutiliza os recursos de GPU, pois pode acontecer de *kernels* maiores serem escalonados na mesma GPU, causando desbalanceamento de carga e, consequentemente, deixando algumas GPUs ociosas.

Na primeira iteração, o algoritmo **MT** percorre cada máquina, na ordem em que aparecem, resolvendo o problema da mochila unidimensional que seleciona os *kernels* a serem escalonados, de acordo com os recursos disponíveis pela GPU. Isto é, para cada máquina é selecionado o número de *kernels* que maximiza o valor da mochila. À medida que os *kernels* vão encerrando, recursos de GPU são liberados, e novos *kernels* são escalonados nestas GPUS. Nessa estratégia o escalonamento dos *kernels* é realizado de forma individual, sem considerar-se a carga de trabalho de cada GPU. Logo, essa estratégia não está livre de um balanceamento de carga ruim.

A heurística min-min (**Min**) distribui os *kernels* entre as GPUs do sistema considerando-se o tempo de conclusão de cada uma delas. Mas depois dessa distribuição, nenhum algoritmo de reordenação é utilizado para tentar otimizar o paralelismo entre os *kernels*.

A quarta estratégia, não só faz a distribuição dos *kernels* entre as GPUs usando a heurística min-min, como também reordena os *kernels* em cada GPU. O algoritmo **MinMT**, une as estratégias **MT** e **Min**. Ele inicia rodando a heurística min-min para distribuir as tarefas entre as máquinas, posteriormente, em cada máquina é executado o algoritmo da mochila a fim de se otimizar o paralelismo entre os *kernels*.

Para simular o ambiente de computação em nuvem, criamos um ambiente com um *datacenter* contendo 2, 4 e 8 *hosts*, sendo que em cada *host* foi alocada uma VM contendo uma única GPU. A seguir apresentamos os resultados da execução dos algoritmos em dois ambientes: ambiente de computação homogênea (GPUs iguais) e ambiente de computação heterogênea (GPUs diferentes).

Ambiente Homogêneo

No ambiente homogêneo, todas as GPUs foram criadas com 32 SMs e configuradas com frequência de processamento 850MHz. Os tempos de execução da simulação do escalonamento em um ambiente homogêneo estão apresentados na Tabela 4.10. Nas colunas estão discriminados o número de *kernels* em cada teste e nas linhas o tempo de execução de cada estratégia para o

número de máquinas virtuais indicado.

Tabela 4.10: Tempos de execução em GPUs com configurações iguais.

		Quantidade de Kernels por VM					
VMs	Algoritmo	8	16	32	64	128	256
2	P	16,51	34,13	68,98	137,28	280,68	615,7
	MT	14,24	26,69	53,31	102,03	223,66	553,11
	Min	15,88	30,81	59,81	117,9	253,11	600,44
	MinMT	14,05	25,22	47,6	91,48	211,86	538,85
4	P	19,73	41,76	85,47	175,98	374,14	838,73
	MT	17,39	37,08	72,59	149,18	297,16	750,59
	Min	18,08	36,04	70,31	148,83	325,34	833,17
	MinMT	16,35	30,96	61,68	124,22	289,75	702,87
8	P	24,01	50,09	105,45	222,53	488,79	1077,13
	MT	20,68	44,8	97,51	198,36	422,35	1046,38
	Min	20,35	42,59	86,77	176,84	449,17	1086,7
	MinMT	19,92	40,06	79,36	154,25	383,53	963,8

O algoritmo padrão (**P**) foi o que apresentou o pior resultado entre os algoritmos testados nesse trabalho. Apesar de haver a distribuição dos *kernels* entre as máquinas, isso é realizado de forma aleatória, sem levar em conta o tempo de execução de cada kernel. Dessa forma, por mais que a quantidade de tarefas seja igual entre as máquinas, o tempo de execução de cada uma pode ser bastante diferente.

A estratégia que utiliza o algoritmo da mochila (**MT**) para distribuir e reordenar os *kernels* apresentou bons resultados, sendo melhor que o min-min (**Min**) em vários cenários, principalmente quando o número de *kernels* é grande.

A distribuição dos *kernels* entre as máquinas e a reordenação deles, são de tarefas não excludentes. Então é possível sugerir que ambas são importantes para resolver o problema, que fica ainda mais evidente ao se avaliar os resultados do algoritmo **MinMT**.

A distribuição dos *kernels* usando o min-min, com a reordenação dos *kernels* nas máquinas em que foram submetidos usando o algoritmo da mochila (**MinMT**), foi a implementação que apresentou os melhores resultados em todos os cenários. Isso significa que o *makespan* é otimizado quando a reordenação de *kernel* acontece após o balanceamento da carga de trabalho entre as máquinas. A Figura 4.9 apresenta o *speedup* do **MinMT** quando comparado com as outras implementações (**P**,**MT**,**Min**).

A vantagem do **MinMT** em relação às outras implementações tende a diminuir quando o número de *kernels* atinge um valor muito grande. Isso acontece porque, apesar do algoritmo da mochila apresentar uma boa solução para o problema, a avaliação dos *kernels* a serem escalonados é realizada de forma individual em cada GPU, além da estimativa de tempo de cada *kernel* não ser

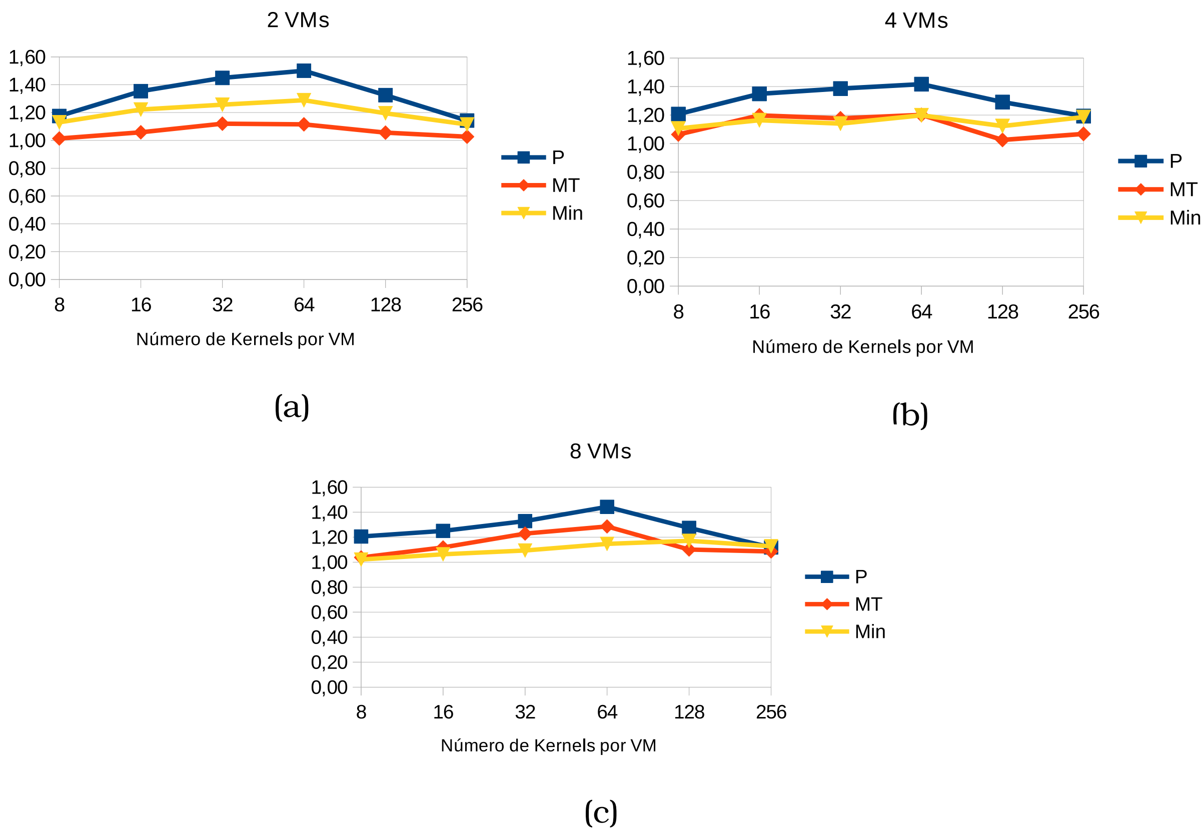


Figura 4.9: *Speedup* da implementação **MinMT** em relação às outras implementações em um **ambiente homogêneo** com: (a) 2 VMs, (b) 4 VMs e (c) 8 VMs.

precisa, pois depende dos SMs livres para cada kernel. Dessa forma, quando aumenta-se o número de *kernels*, o erro da estimativa vai se propagando e interferindo cada vez mais no escalonamento dos próximos *kernels* e, consequentemente, no resultado final do algoritmo.

Ambiente Heterogêneo

No ambiente heterogêneo também configuramos as GPUs com 32 SMs, porém cada GPU possui uma frequência de execução diferente. O poder computacional de cada GPU foi definido conforme a Tabela 4.11 e os tempos de execução da simulação nessas máquinas estão descritos na Tabela 4.12.

A implementação **MinMT** novamente obtém os melhores resultados na maioria dos testes realizados. Quando o ambiente possui 2 VMs e apenas 8 *kernels* é o único caso em que essa implementação não é a melhor. Como são poucos *kernels* a implementação **MT** acaba não sofrendo tanto com a distribuição desbalanceada dos *kernels*, obtendo um resultado melhor, mas apenas 3% superior ao **MinMT**

A heurística min-min nem sempre apresenta bons resultados quando poucos *kernels* estão sendo escalonados. Isso acontece porque essa heurística deixa os *kernels* maiores para o final, estando sujeito a maior impacto fruto

Tabela 4.11: Frequência de execução das GPUs no ambiente heterogêneo.

VM ID	Frequência (MHz)
0	450
1	500
2	550
3	600
4	650
5	700
6	750
7	800

Tabela 4.12: Tempos de execução em GPUs com configurações diferentes.

		Quantidade de Kernels por VM					
VMs	Algoritmo	8	16	32	64	128	256
2	P	214,91	404,72	782,09	1533,65	3024,1	6097,62
	MT	185,18	338,05	650,66	1276,83	2500,42	4999,37
	Min	211,59	387,3	757,51	1515,12	2944,88	6218,11
	MinMT	191,01	329,97	623,29	1190,5	2248,67	4814,52
4	P	229,18	419,56	817,26	1588,95	3159,26	6386,21
	MT	188,13	350,98	675,29	1371,91	2814,58	6173,82
	Min	199,68	365,3	723,5	1466,79	3069,67	6754,7
	MinMT	183,35	317,95	619,96	1224,06	2605,76	5259,29
8	P	223,36	416,06	815,34	1581,12	3183,68	6744,27
	MT	176,61	314,87	630,48	1318,95	2721,44	6467,31
	Min	178,58	313,66	645,03	1370,42	3097,65	7048,04
	MinMT	171,17	289,21	591,5	1186,02	2556,49	5614,65

do desbalanceamento entre as máquinas. É possível observar esse comportamento ao analisar os tempos de execução do min-min (**Min**) para escalonar 8, 16, 32 e 64 *kernels* com 2, 4 e 8 máquinas. Os tempos de execução diminuem à medida que o número de máquinas aumenta, mesmo que a proporção máquina/*kernel* se mantenha constante. Nestas execuções os *kernels* foram escalonados à proporção de 8 *kernels* por máquina. Por exemplo, para 4 máquinas, 32 *kernels* $(8\dot{4} = 32)$ serão distribuídos entre elas.

A estimativa dos tempos de execução dos *kernels* é de grande importância para a eficácia das estratégias **Min**, **MinMT** e **MT**, principalmente, quando se lida com uma grande quantidade de *kernels*. Como já discutido, o tempo estimado dessas estratégias não é exato, por isso, com o aumento no número de *kernels*, o *speedup* dessas estratégias em relação à padrão tende a diminuir. A Figura 4.10 apresenta os *speedups* da estratégia **MinMT** que foi a que apresentou os melhores resultados, quando comparada com as outras estratégias.

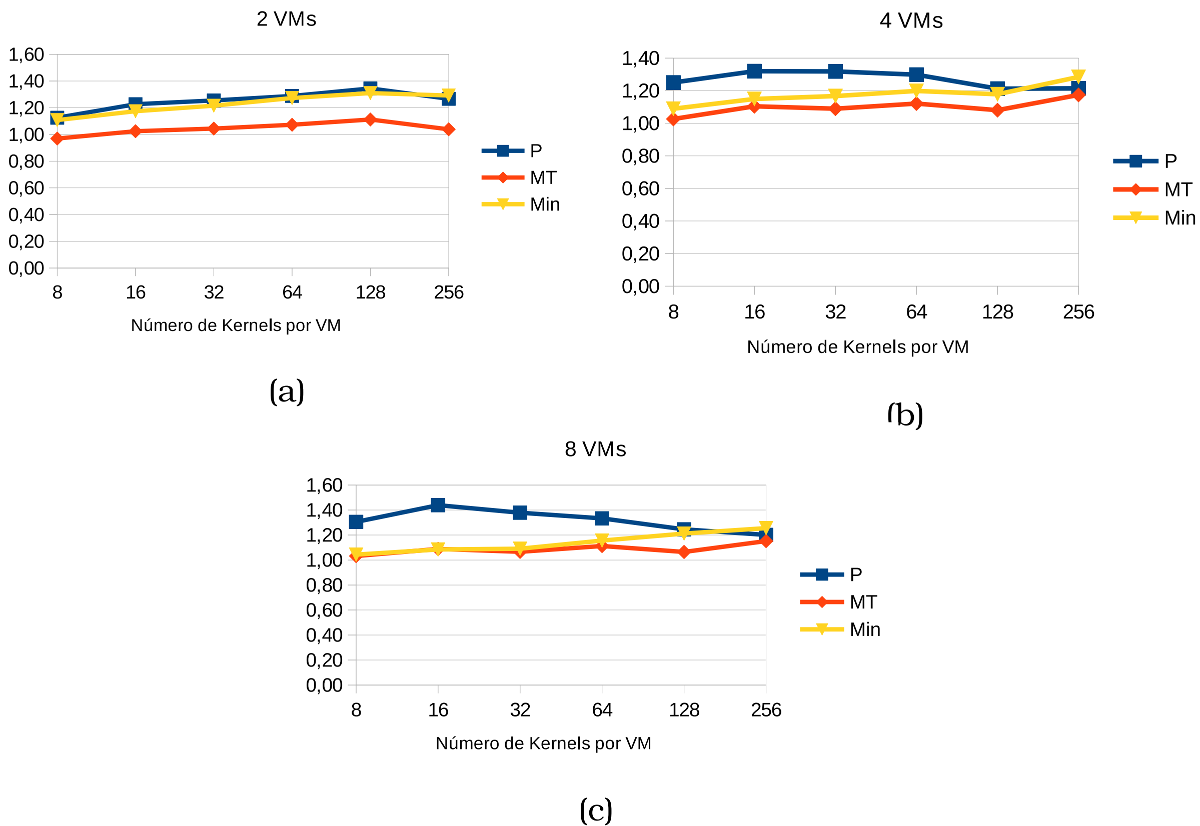


Figura 4.10: *Speedup* da implementação **MinMT** em relação às outras implementações em um **ambiente heterogêneo** com: (a) 2 VMs, (b) 4 VMs e (c) 8 VMs.

4.4 Conclusões e Trabalhos Futuros

Nesse capítulo apresentamos a comparação de algoritmos de escalonamento de tarefas em ambientes heterogêneos e a reordenação de *kernels* em ambientes simulados de computação em nuvem.

Considerando os tempos obtidos, fazer um escalonamento prévio das tarefas utilizando uma das heurísticas max-min ou min-min traz ganhos consideráveis para os resultados no Cloudsim. Visto que tratam-se de heurísticas simples, que se resolvem muito rapidamente, a adição delas como um escalonador nativo da ferramenta pode facilitar a definição das estratégias pelo usuário.

Além disso, foi identificado que em um ambiente consistente, as heurísticas apresentam tempos de conclusão semelhantes sempre que a relação tarefas/máquinas é a mesma, e também quando o poder computacional é o mesmo para um determinado número de tarefas, mesmo que a quantidade de máquinas seja diferente. Isso é importante porque sugere ao usuário que, ao contratar um ambiente de computação em nuvem, ele possa escolher um ambiente com mais ou menos máquinas e máquinas mais complexas ou mais simples, sem afetar de forma significativa o resultado final.

Os testes realizados para resolver o problema de reordenação de *kernels* em GPUs consistiram em três configurações: uma única GPU, várias GPUs iguais e várias GPUs diferentes. A reordenação de *kernels* em uma única GPU se mostrou eficiente no ambiente simulado, o que motivou os testes utilizando múltiplas GPUs, considerando ambientes homogêneos (GPUs iguais) e heterogêneos (GPUs diferentes). Em todos eles o tempo de conclusão das tarefa (*makespan*) é otimizado pelas reordenações.

Dessa maneira, a heurística min-min (**Min**) apresentou-se como uma boa alternativa no escalonamento de *kernels* na GPU, mas foi ainda melhor quando empregada junto com o algoritmo da mochila (**MT**). Isto é, a estratégia que se mostrou mais eficiente para efetuar a reordenação no ambiente simulado de computação em nuvem foi distribuir as tarefas entre as máquinas utilizando o min-min, e depois reordená-las dentro das máquinas usando a mochila (**MinMT**).

O ambiente simulado deste capítulo é uma simplificação do ambiente real, pois, por exemplo, divergem no número de filas da GPU e nas dimensões dos recursos de GPU disponíveis. Enquanto no ambiente simulado só existe uma fila, que permite a execução paralela dos *kernels*, e a utilização dos recursos da GPU ocorrem em uma única dimensão, no ambiente real são três dimensões e existem até 32 filas, que devem ser utilizadas para estabelecer a independência entre os *kernels* e afetam a ordem de execução deles. Dessa forma, a solução desse problema em um ambiente real torna-se muito mais difícil. Como o ambiente simulado apresentou resultados promissores, realizamos estudos usando GPUs em ambientes reais, que são o objeto do próximo capítulo.

CAPÍTULO

5

Reordenação de Kernels em GPUs

Enquanto as CPUs atingiam seu limite de processamento sequencial, as GPUs cresciam exponencialmente em desempenho devido seu alto poder de processamento paralelo (Brodtkorb et al., 2013). O paralelismo por meio de *threads* era o principal meio de paralelismo empregado nesses dispositivos. Esse método de paralelismo foi bastante usado para explorar o hardware da GPU e esconder latências, como acesso à memória global, divergência no fluxo de controle, ou sincronização.

Apesar do sucesso do paralelismo por meio de *threads* em prover *speedup* substancial em várias aplicações, as GPUs mais novas são capazes de executar os *kernels* de forma concorrente (Cruz et al., 2019). Kernel é uma função que é executada em uma GPU. A chamada de um kernel consiste em passar as configurações de *threads* e memória compartilhada, que serão usadas na execução da função pela GPU.

Ao utilizar execução concorrente de *kernels*, os programadores podem especificar a fila de execução de cada kernel, colocando-os em diferentes *streams* CUDA (Cruz et al., 2019), e sempre que houver recursos de hardware disponíveis, *kernels* em diferentes filas de execução poderão executar em paralelo na mesma GPU. Dessa forma, a ordem em que esses *kernels* são submetidos, fará total diferença no tempo de execução e na taxa de utilização da GPU.

Neste capítulo, apresentamos a formalização do problema de reordenação de *kernels* na GPU e os resultados obtidos com algoritmos de reordenação, afim de tentam maximizar a taxa de utilização dos recursos da GPU, por meio da tecnologia Hyper-Q.

5.1 Arquitetura de uma GPU

As Unidades de Processamento Gráficos (GPUs) são unidades de processamento especializadas que foram inicialmente concebidas com o propósito de acelerar operações com vetores, como a renderização gráfica. Com o avanço dessa tecnologia, as GPUs tornaram-se unidades de processamento paralelo de propósito geral (Amarís et al., 2015). Uma plataforma de computação paralela que podemos citar é o CUDA (*Compute Unified Device Architecture*). O CUDA facilita o desenvolvimento de algoritmos em GPUS e foi introduzido pela NVIDIA em 2006 em suas placas de processamento gráfico (Amarís et al., 2015).

Grande quantidade de espaço dos transistores da CPU são dedicados ao paralelismo de instrução como pipeline, predição de desvios, execução especulativa, execução fora de ordem, deixando apenas uma pequena fração para unidades de execução de inteiros e de ponto flutuante. Por outro lado, as GPUs são compostas por centenas de *cores* simples que lidam com milhares de *threads* concorrentes, projetadas a maximizar a vazão de execução de inteiro e ponto flutuante (Brodtkorb et al., 2013).

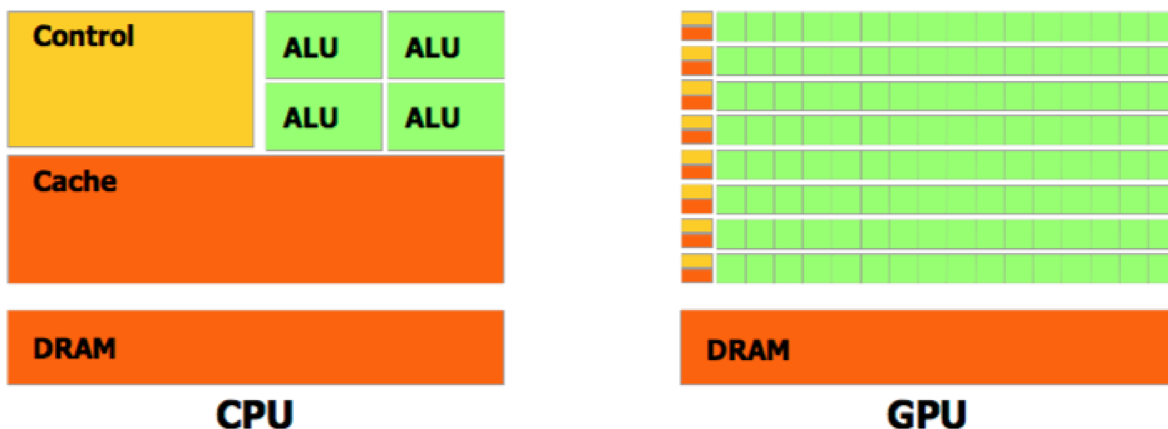


Figura 5.1: Diferença entre a arquitetura de uma CPU e de uma GPU (**?**).

Há algum tempo temos visto uma rápida adoção de unidades de processamento gráficos (GPU) para computação de alto desempenho (Wu et al., 2013). Apesar de o paralelismo parecer um excelente meio para melhorar o desempenho, não basta aumentar o paralelismo para melhorá-lo, problema frequentemente referenciado como Lei de Amdahl (Amdahl, 1967). Muitas vezes, novos algoritmos devem ser propostos para atingir o nível de paralelismo desejado, além de estarmos limitados pelo número de núcleos do dispositivo de hardware sendo utilizado. Portanto, os programadores precisam primeiro identificar as partes paralelas do problema e analisar as diversas soluções a fim de identificar a que melhor se encaixa no problema em questão.

Como uma arquitetura paralela, as GPUs aceleram significativamente muitas aplicações paralelas regulares. Mas seu benefício para aplicações irregulares é bem menos substancial, especialmente quando a aplicação contém acesso dinâmico e irregular à memória (Wu et al., 2013). Por isso, entender apenas o algoritmo nem sempre é suficiente para justificar um resultado, sendo necessária uma análise de como o hardware interage com os comandos gerados pelo software.

A arquitetura divide o dispositivo em *grids*, blocos e *threads* em uma estrutura hierárquica como mostrado na Figura 5.2. Uma grid é um grupo de blocos de *threads* que executam o mesmo kernel. Cada bloco contém um grupo de *threads* que podem ser sincronizadas entre si, além de uma quantidade de memória que pode ser compartilhada pelas *threads* que pertencem a esse bloco. *Threads* de blocos diferentes não podem ser sincronizadas, a não ser pelo término do kernel que estão executando. Todos os blocos de uma grid rodam o mesmo código (Ghorpade et al., 2012).

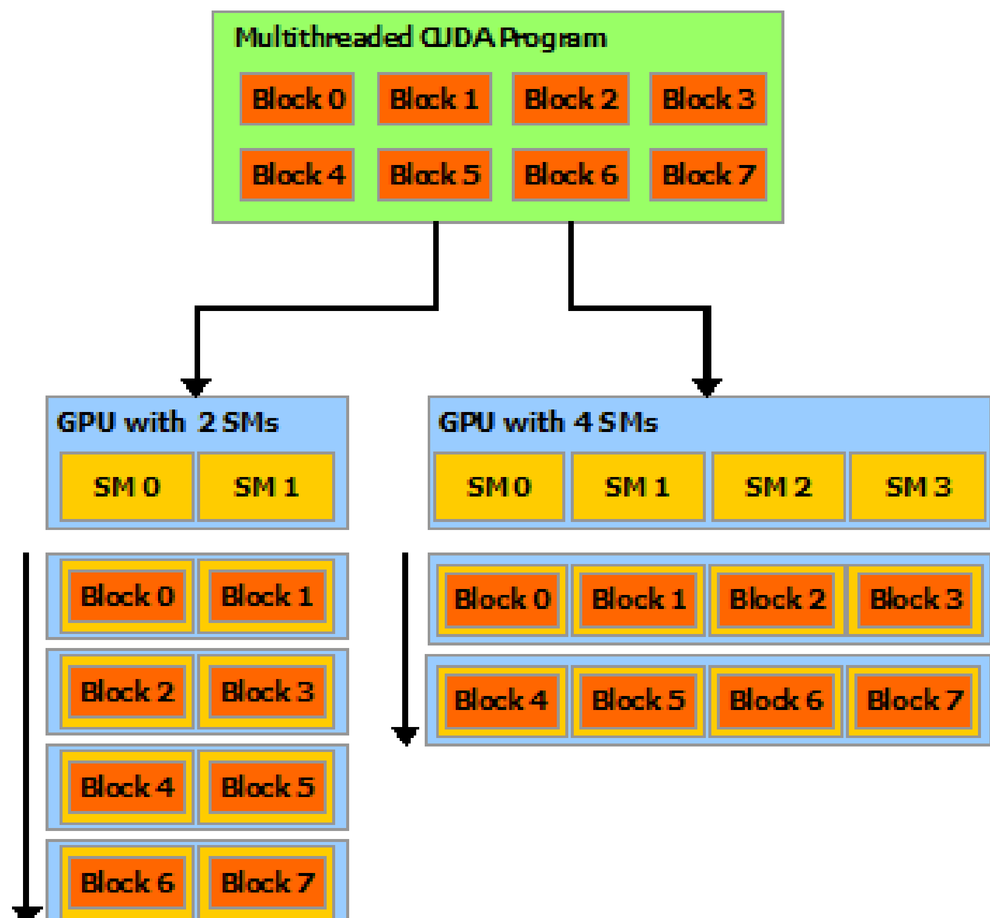


Figura 5.2: Os blocos de *threads* podem ter até 3 dimensões e são distribuídos entre os SMs da GPU.

As unidades de processamento gráfico (GPUs) são consideradas arquiteturas SIMT (*Single Instruction, Multiple Thread*). Nesse modelo, múltiplas *thre-*

ads executam concorrentemente usando uma única instrução (Clua e Zamith, 2015). No caso das GPUs, o grupo de *threads* que executam juntas a mesma instrução é chamado de *warp*. Um *warp* possui 32 *threads* e podem haver vários *warps* executando concorrentemente entre os multiprocessadores (*streaming multiprocessor - SM*) da GPU. As GPUs possuem vários SMs (Chang et al., 2016) que podem executar em paralelo. Cada SM é composto por centenas de *cores* e é capaz de manter ativos vários blocos e *warps* ao mesmo tempo.

Apesar de as GPUs possuírem uma largura de banda de memória bastante superior às de uma CPU, elas ainda levam muitos ciclos do *clock* para iniciar a busca de um elemento na memória. Essa latência é superada pela GPU por meio de uma rápida troca entre *threads*. Ao ocorrer um acesso à memória por uma *thread*, outra é imediatamente despachada para execução (Brodtkorb et al., 2013). Portanto, a única forma de esconder a latência é executando *warps* suficientes para manter o hardware completamente ocupado. Essa métrica é chamada de ocupação e é medida em percentual da utilização dos recursos. A ocupação mede o número de *warps* executando concorrentemente em um multiprocessador dividido pelo número máximo de *warps* que pode ser executado concorrentemente. Dessa forma, o paralelismo de *kernels* surge como uma forma de melhorar essa métrica, tendo em vista que mais *kernels* estarão executando concorrentemente na mesma GPU.

5.2 Maximizando a Ocupação da GPU

Otimizar a ocupação da GPU é uma forma de melhorar a sua eficiência, já que a latência de um acesso à memória pode ser camuflado pela execução de outra *thread*. Nesta seção apresentamos alguns trabalhos que visavam maximizar essa ocupação por meio da concorrência entre *kernels*.

Guevara et al. (2009) propuseram unir *kernels* que subutilizavam os recursos de processamento da GPU com o intuito de rodá-los concorrentemente na GPU NVIDIA. Essa fusão é realizada em tempo de compilação e mostrou aumentar o *throughput* em todos os casos em que a GPU era subutilizada por um único kernel. De maneira semelhante, Gregg et al. (2012) propõem um escalonador de kernel que une diferentes *kernels* de aplicações OpenCL para rodarem em um único dispositivo e identificaram que, apesar de melhorar a utilização da GPU, em alguns casos piora o desempenho do kernel. Wang et al. (2010) propuseram vários métodos de fusão de kernel para reduzir o consumo de energia e melhorar a eficiência energética. Por meio da fusão de *kernels* conseguiram melhor utilização e maior balanceamento de demanda dos recursos de hardware.

Peters et al. (2010) superaram as limitações de concorrência de kernel entre aplicações diferentes criando um kernel persistente que consiste em um conjunto de funções, que são chamadas pelos usuários e podem ser executadas concorrentemente. Com essa contribuição a GPU pode ser compartilhada entre vários clientes, aumentando a flexibilidade sem perda de performance.

Outros autores tentaram explorar a granularidade dos *kernels* a fim de obter mais oportunidades de compartilhamento no tempo. Zhong e He (2013) apresentaram uma estratégia de divisão dos *kernels*, chamada *Kernelet*. O Kernelet divide um kernel da GPU em múltiplos sub-*kernels* e, por meio do gerenciamento de memória e da transferência de dados, escalona-os para execução na GPU. Ravi et al. (2011) apresentam um framework que permite que aplicações executem em máquinas virtuais e compartilhem uma ou mais GPUs. Eles estendem um virtualizador de GPU *open source*, para incluir o compartilhamento eficiente da GPU, e testaram oito *kernels* em duas fermi GPUs.

Tanasic et al. (2014) propuseram um conjunto de extensões de hardware para permitir cargas de trabalho concorrentes na GPU. Modelaram dois mecanismos de preempção e desenvolveram uma política de escalonamento que dinamicamente distribui os *cores* da GPU entre os *kernels* executando concorrentemente, de acordo com suas prioridades.

Inicialmente, apenas *warps* de um mesmo kernel poderiam ser executados concorrentemente, agora, as novas gerações das GPUs NVIDIA implementam execução concorrente de *kernels* (Cruz et al., 2019), por meio da tecnologia Hyper-Q. A tecnologia Hyper-Q permite que blocos de *threads* de diferentes *streams* compartilhem os mesmos recursos da GPU. Cada *stream* representa um fila que executa uma sequencia de *kernels* na ordem FIFO (First In First Out) (Breder, 2016). Assim, o kernel k_{i+1} só poderá ser executado quando o kernel k_i terminar. Entretanto, se esses *kernels* estiverem em diferentes *streams*, poderão executar concorrentemente caso haja recursos disponíveis (Pai et al., 2013).

Na arquitetura Kepler, a primeira a implementar a tecnologia Hyper-Q, a GPU possui 32 filas de hardware, não tendo limite bem definido de filas de *streams*. Dessa forma, quando são criadas mais *streams* do que a quantidade de filas de hardware, a GPU compartilha mais de uma *stream* numa mesma fila de hardware (NVIDIA, 2012; Pai et al., 2013). Mas para a execução paralela dos *kernels* é necessário que hajam recursos de GPU disponíveis, como memória compartilhada, registradores e *threads*. Por exemplo, considere uma GPU hipotética com um SM contendo 1024 registradores, 8KB de memória compartilhada e o número máximo de *threads*, 1024. Considere também um kernel k_i com 2 blocos de *threads*, cada bloco contendo 256 *threads* e usando

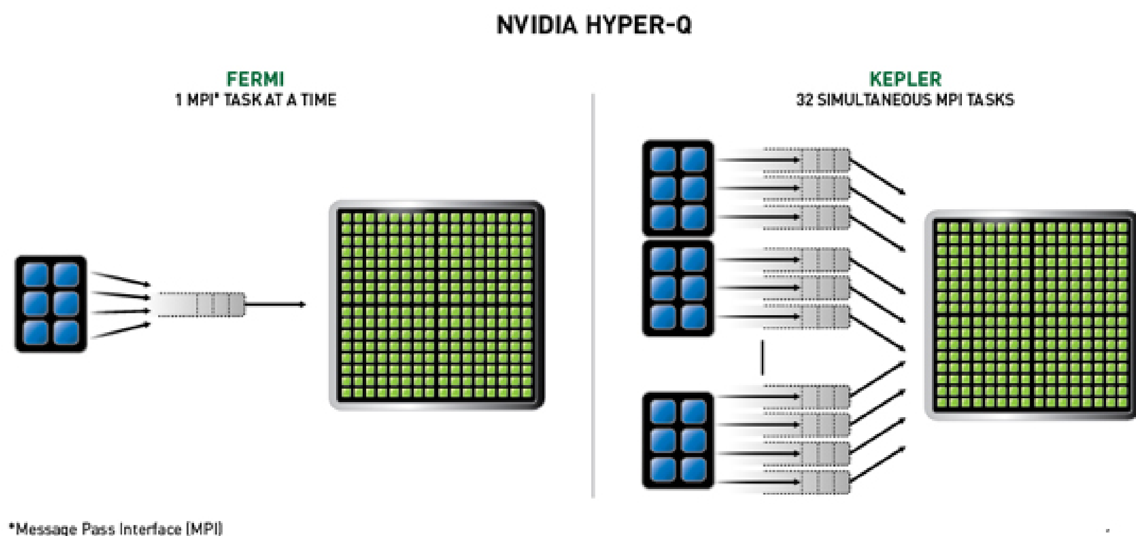


Figura 5.3: A concorrência da Fermi é limitada a uma única fila de trabalho. Na Kepler, as *streams* podem executar concorrentemente usando filas separadas de tarefas (Breder, 2016).

2KB de memória compartilhada, e cada *thread* usando 1 registrador. Neste cenário, o SM terá recursos não utilizados: metade do máximo de *threads*, metade da memória compartilhada e metade do número de registradores. Esses recursos não utilizados podem ser alocados a outros blocos de *threads* de diferentes *kernels* (desde que alocados em diferentes *streams*).

5.3 Reordenação de Kernels

A descrição do problema, conforme descrito no trabalho de Breder (2016), consiste das seguintes características. Dada uma lista de *kernels* K = $k_0, k_1, ..., k_{N-1}$ a serem submetidos para execução em uma GPU D (e que k_i é submetido antes de k_{i+1}). Considere que a GPU possui um número de SMs igual a NSM e que os recursos de cada SM sejam representados por $R_j \mid j \in 0,1,2$.

- R_0 representa o tamanho da memória compartilhada de cada SM.
- R_1 representa o número máximo de registradores de cada SM.
- R_2 representa o número máximo de *threads* suportado 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 momento durante a execução, existem m *kernels* sendo executados concorrentemente em D, usando um percentual de W_j. Os recursos disponíveis neste instante são chamados $W_j^{av} \mid j \in 0,1,2$. Toda vez que um kernel termina e libera recursos, novos *kernels* são selecionados para

execução, de acordo com a quantidade de recursos disponíveis. Para cada kernel ainda não submetido, as seguintes informações são coletadas [1]:

- t_i^{est}: o tempo estimado de execução [2].
- sh_i: a quantidade de memória compartilhada requerida por bloco.
- nr_i: o número de registradores requeridos por bloco.
- nt_i: o número máximo de *threads* lançadas por bloco.
- nb_i: o número de blocos.

Para cada kernel k_i foi atribuído um peso w_ij ($i \in 0,...,N-1,\ j \in 0,1,2$) que representa a quantidade de recursos necessários j: $w_{i0} = sh_i \times nb_i,\ w_{i1} = nr_i \times nb_i$, $w_{i2} = nt_i \times nb_i$. Também foi atribuído a cada kernel k_i um valor v_i calculado pela média do percentual de recursos requeridos por unidade de tempo, descritos pela Equação 5.1. *Kernels* com tempos de execução menores terão maiores valores de v_i.

$$v_i = \frac{\frac{sh_i}{R_0} + \frac{nr_i}{R_1} + \frac{nt_i}{R_2}}{3t_i} \tag{5.1}$$

O problema da mochila consiste em encontrar uma quantidade de *kernels* de tal forma que maximize a soma de seus valores, sem ter a soma de seus pesos excedendo o valor de W. O problema da mochila 0-1 é formulado como a seguinte maximização:

$$\begin{aligned} \text{maximizar} \sum_{i=0}^{N-1} v_i x_i \\ \text{sujeito a} \sum_{i=0}^{N-1} w_{ij} x_i \leq W_j | j \in 0,1,2 \end{aligned} \tag{5.2}$$

em que $x_i \in 0,1,\ x_i = 1$ se o item i deveria ser incluído na mochila, e $x_i = 0$ caso contrário. Neste problema específico, a maximização do valor da mochila representa a maximização do uso dos recursos da GPU.

5.4 Implementações

Nesta seção, descrevemos três estratégias de reordenação de kernel implementadas neste trabalho: padrão (**P**), fit (**F**) e mochila com linearização do peso (**ML**). A duas primeiras são mais simples e não fazem avaliações de diferentes ordens de execução, enquanto a última é baseada no problema da mochila

[1] As quantidades de memória compartilhada, registradores e *threads* podem ser dinamicamente obtidas por meio da ferramenta da NVIDIA.

[2] Existem várias abordagens de predição de tempo de execução baseados em diferentes técnicas, porém este estudo está fora do escopo deste trabalho.

unidimensional, o peso é obtido por meio da linearização de três recursos de GPU: memória compartilhada, *threads* e registradores.

A implementação padrão (**P**) consiste em submeter um kernel para cada *stream* de execução da GPU e repetir o processo até que todos os *kernels* sejam submetidos a algum *stream*. Nesta estratégia, não se levam em consideração os recursos de GPU disponíveis nem os recursos necessários para cada kernel. Os *kernels* são simplesmente distribuídos igualmente entre as *streams* disponíveis. Chamamos essa estratégia de *padrão*, e é descrita pelo Algoritmo 9.

Algoritmo 9 Pseudocódigo da implementação de escalonamento padrão (**P**).

1: $j \leftarrow 0$
2: **for** `i = 0, n-1` **do**
3: $\quad k_i \leftarrow \text{kernels}[i]$
4: $\quad \text{enfileira}(k_i, j)$
5: $\quad j \leftarrow (j+1) \ \% \ 32$

As próximas duas implementações executam os passos do Algoritmo 10, com a única diferença de executarem diferentes algoritmos de seleção de *kernels* (passos 1 e 11). O passo 1, consiste da primeira seleção de *kernels* a rodar na GPU. Como inicialmente não há nenhum kernel executando, todos os recursos da GPU estão disponíveis e as filas de execução estão todas vazias. Dessa forma, os *kernels* inicialmente selecionados são escalonados um em cada *stream* da GPU (passos 4, 5 e 6). Depois, conforme os primeiros *kernels* forem finalizando (passo 9) e liberando recursos da GPU (passo 10), novos *kernels* são selecionados (passo 11). Estes novos *kernels* são escalonados (passo 14) nas *streams* que possuem tempo estimado de finalização menores (passo 15), isto é, aquelas que vão finalizar seus *kernels* primeiro.

Os passos de seleção da implementação *Fit* estão descritos no Algoritmo 11. A estratégia consiste em escolher *kernels* que caibam inteiros na GPU ao mesmo tempo (passos 3 ao 6), fazendo a verificação na ordem de seus valores (passo 2), calculados de acordo com a Equação 5.1. Quando não houver mais *kernels* que caibam inteiros, é verificado se ao menos um bloco CUDA de algum kernel cabe na GPU, aquele que couber é escalonado junto (passos 9 ao 11). Quando um kernel termina, o processo de escalonamento é iniciado novamente, porém apenas com os recursos de GPU (W^{av}) liberados por esse kernel.

A estratégia baseada na mochila, que efetua a linearização dos pesos (**ML**), foi baseada na implementação do trabalho de Breder (2016). Essa implementação leva em conta três recursos de GPU para fazer o escalonamento: memória compartilhada, número de *threads* e número de registradores. Como não é conhecido uma solução ótima do problema da mochila multidimensional em

Algoritmo 10 Pseudocódigo das implementações de escalonamento: **G**, **F** e **ML**.

1: $selectedKernels \leftarrow$ AlgoritmoDeSelecaoDeKernels($kernels, W^{av}$)
2: $j \leftarrow$
3: **for** `i = 0, size(selectedKernels)` **do**
4: $k_i \leftarrow selectedKernels[i]$
5: enfileira(k_i, j)
6: $j \leftarrow (j+1)$ % 32
7: RemoveKernels($selectedKernels, kernels$)
8: **while** `kernels != NULL` **do**
9: $j \leftarrow$ StreamDoKernelFinalizado()
10: $W^{av} \leftarrow$ PesoLiberadoPeloKernelFinalizado()
11: $selectedKernels \leftarrow$ AlgoritmoDeSelecaoDeKernels($kernels, W^{av}$)
12: **for** `i = 0, size(selectedKernels)` **do**
13: $k_i \leftarrow selectedKernels[i]$
14: enfileira(k_i, j)
15: $j \leftarrow$ StreamComMenorTempoDeFinalizacao()
16: RemoveKernels($selectedKernels, kernels$)

Algoritmo 11 Pseudocódigo da seleção de *kernels* fit (**F**).

1: **procedure** ALGORITMODESELECAODEKERNELS(*kernels*,W^{av})
2: $kernelsSorted \leftarrow$ SortByValue($kernels$)
3: **for** `i = 0, size(`$kernels$`)` **do**
4: **if** totalWeight($kernels[i]$) $<= W^{av}$ **then**
5: $selectedKernels$.add($kernels[i]$)
6: $W^{av} \leftarrow W^{av} -$ totalWeight($kernels[i]$)
7: **for** `i = 0, size(`$kernels$`)` **do**
8: **if** notSelectedYet($kernels$) **then**
9: **if** OneBlockWeight($kernels[i]$) $<= W^{av}$ **then**
10: $selectedKernels$.add($kernels[i]$)
11: $W^{av} \leftarrow W^{av} -$ totalWeight($kernels[i]$)

tempo polinomial, eles simplificaram o problema por meio da linearização dessas três medidas e resolveram o problema como se fosse uma mochila clássica 0-1. Dessa forma, os recursos da GPU foram linearizados para definir o peso suportado pela mochila e os recursos necessários para a execução de cada kernel também foram linearizados para identificar se o kernel cabia ou não na mochila. O cálculo dos pesos dos *kernels* e da mochila é considerado como sendo linear, mas sempre que um kernel é selecionado, os recursos de cada dimensão são subtraídos individualmente. Essa estratégia está representada pelo Algoritmo 12.

Algoritmo 12 Pseudocódigo da seleção de *kernels* usando o algoritmo da mochila (**ML** e **MT**).

```
1: procedure AlgoritmoDeSelecaoDeKernels(kernels,W^av)
2:     for j = 0, W^av do
3:         M[0, j] ← 0
4:     for i = 1, size(kernels) do
5:         for j = 1, 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])
```

5.5 Resultados Experimentais

Neste trabalho realizamos vários experimentos e foram avaliadas várias métricas de desempenho em relação ao paralelismo da GPU a nível de kernel, isto é, foi avaliado o comportamento dos *kernels* ao serem distribuídos em diferentes filas de execução da GPU.

Como ter o controle da utilização dos recursos da GPU pelos *kernels* era fundamental para a realização dos testes desse trabalho, foram utilizados *kernels* sintéticos, ou seja, *kernels* criados especificamente para esse trabalho. Esses *kernels* foram gerados por meio de uma adaptação do código do trabalho de Breder (2016) e foram utilizadas as mesmas dimensões de *kernels* e blocos.

As quantidades de *kernels* testados foram 32, 64, 128 e 256. O número de blocos foi gerado aleatoriamente de acordo com o limite máximo daquele caso de teste. Os limites de blocos utilizados foram 32, 64, 128, 256 e 512 blocos, para cada uma das quantidades de *kernels*. Dessa forma, para um limite de blocos igual a 32, um número aleatório entre 1 e 32 foi gerado para cada kernel, para o limite de blocos 64, um número aleatório entre 1 e 64 foi gerado, e assim sucessivamente. Isto é, o par número de *kernels* e quantidade de blocos definem um caso de teste. O número de *threads* de cada bloco

também foi gerado de forma aleatória, podendo ser 32, 64, 128, 256, 512 ou 1024, independente do caso de teste. Da mesma forma, o tamanho da memória compartilhada de cada bloco independe do caso de teste e foi limitada ao tamanho máximo de um SM da GPU.

Além disso, foram geradas 3 distribuições de *kernels* diferentes em relação ao número de blocos para cada uma das dimensões, seguindo os coeficientes de distribuição 20%, 80% e 100%. As distribuições foram realizadas da seguinte maneira: 20% de *kernels* pequenos e 80% de *kernels* grandes; 80% de *kernels* grandes e 20% de *kernels* pequenos; e por último 100% que considera todos os valores sendo gerados aleatoriamente. Kernels pequenos e grandes foram definidos de acordo com o número máximo de blocos da dimensão sendo gerada. Os *kernels* pequenos foram definidos como aqueles que possuem no máximo 25% do total de blocos máximo, e os *kernels* grandes foram definidos como sendo os *kernels* que possuem pelo menos 75% do total de blocos máximo. Por exemplo, quando o número máximo de blocos está definido como 128 *kernels*, os *kernels* pequenos terão no máximo 32 blocos (25% de 128), e os *kernels* grandes terão pelo menos 96 blocos (75% de 128).

Para cada dimensão do problema foram gerados 5 conjuntos de *kernels* diferentes, no qual 5 execuções foram realizadas para cada um deles, totalizando 25 execuções para cada dimensão do problema. A média e o desvio padrão foram calculados e apresentados como resultado dos testes.

As métricas avaliadas neste capítulo, foram:

- **Tempo de execução**: O tempo de execução é contabilizado a partir do momento em que o primeiro kernel inicia até o instante em que o último kernel termina. Para comparar essa métrica entre as estratégias, foi utilizado o speedup. O tempo de execução da versão otimizada é dividido pelo tempo de execução da versão sem otimização.

- **Tempo de resposta**: O tempo de resposta é medido a partir do momento que o conjunto de todos os *kernels* inicia até o instante em que o kernel sendo medido termina. Nesta avaliação, foi mensurada a média dos tempos de resposta de cada kernel e foi calculada a proporção de melhora das versões paralelas em relação à versão sequencial. Isto é, a média dos tempos de resposta da versão sequencial foi dividida pela média dos tempos de resposta da versão paralela.

- **Tempo de Duração**: O tempo de duração é calculado de forma individual para cada kernel e mede o instante que um kernel inicia sua execução na GPU até o instante que ele termina. O objetivo é identificar o impacto da execução dos *kernels* em múltiplas filas, quando comparado com a execução em uma única fila, de forma sequencial. Para isso, foram medidos

o tempo de duração TT_i^{Alone} de cada kernel, quando executado sozinho na GPU e o tempo de duração TT_i de cada kernel, quando executado de forma concorrente com os outros *kernels*. A fórmula utilizada para calcular essa métrica é dada por:

$$TD = \frac{1}{N}\sum_{i=1}^{N}\frac{TD_i}{TD_i^{Alone}}$$

- **Consumo Energético**: O consumo energético foi medido utilizando a ferramenta Data Center GPU Manager NVIDIA (2019), pela qual foram coletados a média dos watts durante a execução dos *kernels*. Além disso, o consumo em joules também foi calculado, utilizando a fórmula:

$$J = w \cdot t$$

em que J é o consumo em joules, w são os watts e t é o tempo de execução.

5.6 Análise das Estratégias em Relação ao Número de Filas

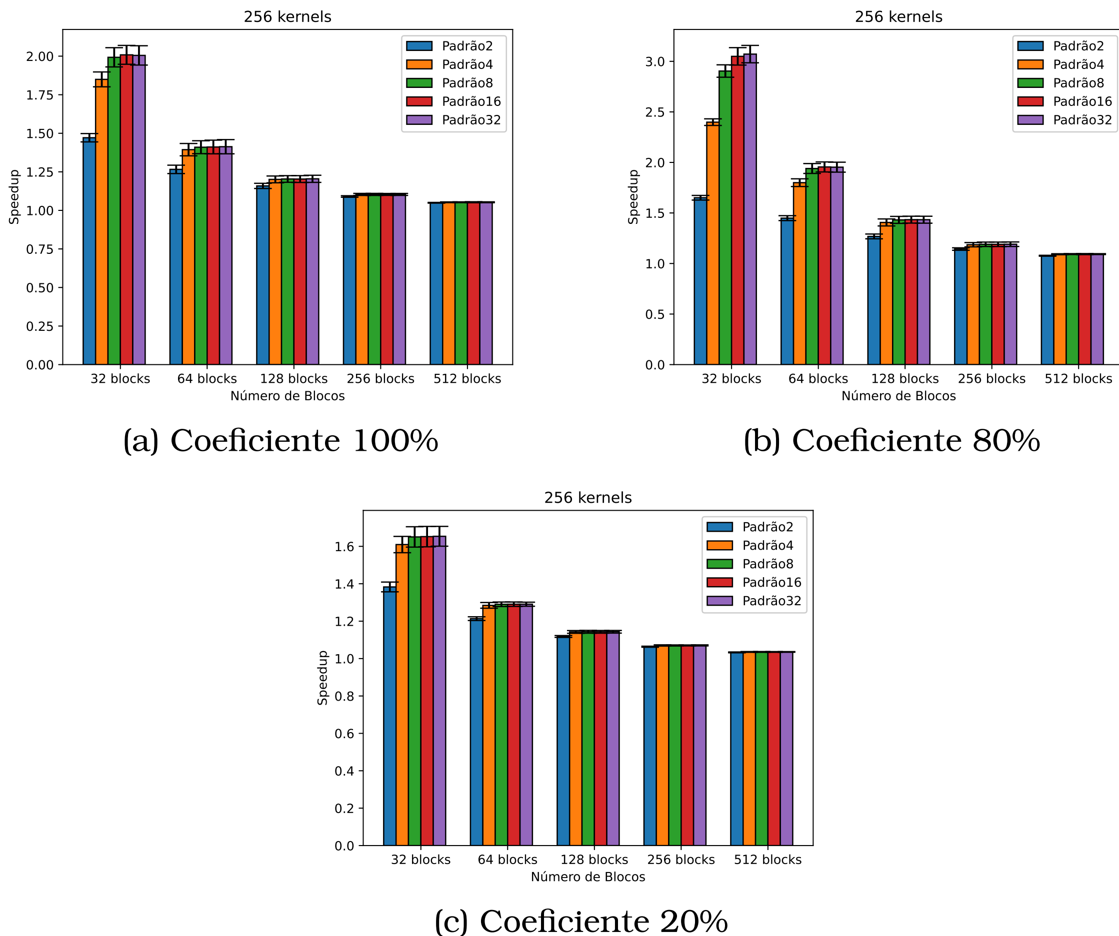
Nesta seção, as estratégias são avaliadas individualmente quanto ao número de filas da GPU utilizadas. O objetivo é avaliar o comportamento de cada estratégia quando se aumenta o número de filas da GPU. Para isso, foram utilizadas seis quantidades de filas diferentes: 1, 2, 4, 8, 16 e 32.

5.6.1 Análise da Estratégia Padrão (**P**)

Nesta seção, avaliamos o impacto do número de filas da GPU nas métricas da estratégia padrão (**P**), que distribui igualmente os *kernels* entre as filas de execução da GPU e os executa na ordem em que chegaram.

Tempo de Execução

Os resultados dos testes com 32, 64, 128 e 256 *kernels* tiveram resultados muito semelhantes, alterando apenas o desvio padrão, que nas estratégias menores acabou sendo maior. Isso é esperado, pois execuções mais rápidas tendem a ser mais afetadas pelo *overhead* da chamada da função. Sendo assim, optou-se por apresentar os resultados somente das execuções com 256 *kernels*. A Figura 5.4 apresenta os *speedups* dos tempos de execução das estratégias de acordo o número de filas para os coeficientes 100%, 80% e 20%.



(a) Coeficiente 100%

(b) Coeficiente 80%

(c) Coeficiente 20%

Figura 5.4: *Speedup* do tempo de execução paralelo em relação ao tempo de execução sequencial para 256 *kernels*, usando a estratégia padrão.

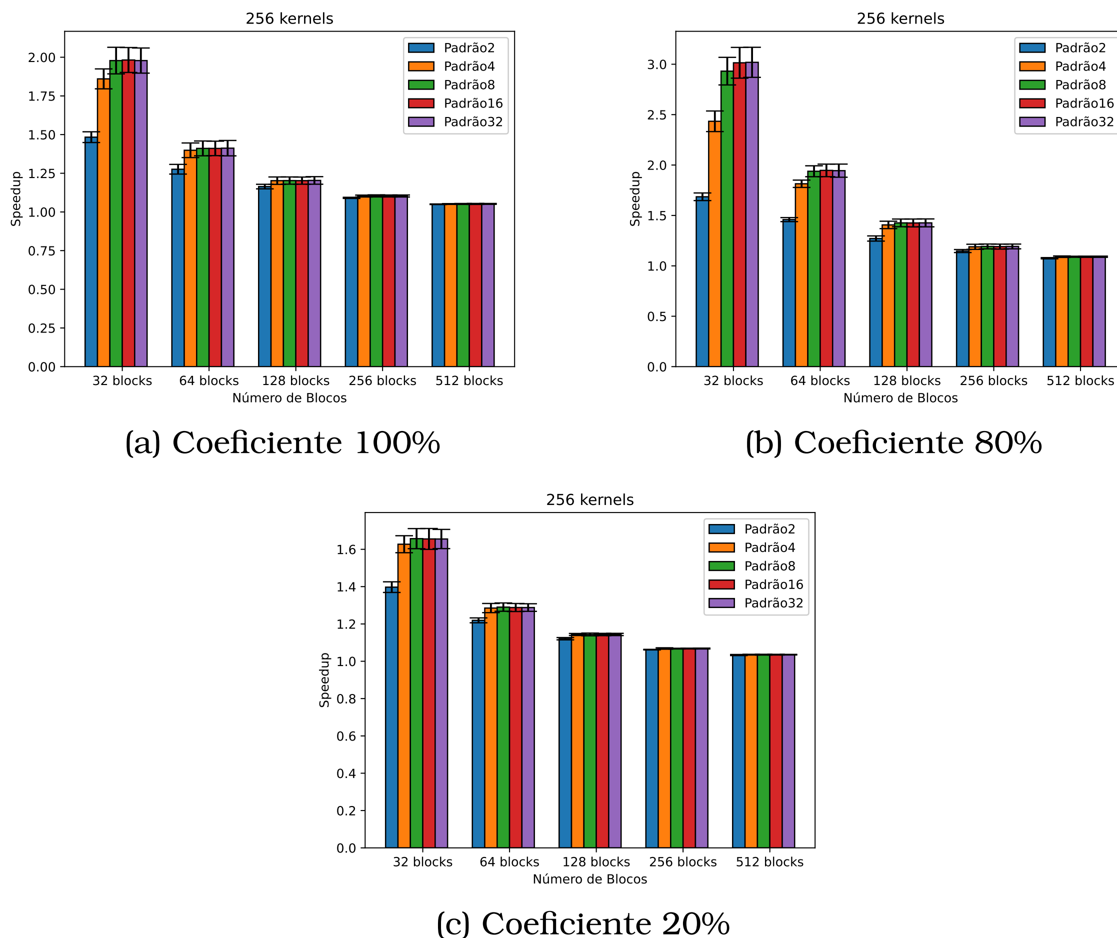
A Figura 5.4(a) mostra que o aumento no número de filas tende a melhorar o tempo de execução, porém, a partir de 8 filas, o tempo de execução é quase constante, o que sugere que utilizar todas as filas de execução nem sempre vai significar melhor desempenho. Além disso, podemos notar que com o aumento do tamanho dos blocos, o tempo de execução ao se adicionar mais filas muda muito pouco. Isso já era esperado, pois *kernels* grandes não permitem que outros *kernels* rodem juntos com eles. Dessa forma, a melhora no tempo de execução se dá pelo camuflagem do *overhead* da chamada dos *kernels* ao utilizar mais filas de execução e pelos *kernels* menores que podem rodar concorrentemente.

Os coeficientes que definem o número de blocos de um kernel também tem papel relevante no *speedup* dos tempos de execução. Enquanto os testes com distribuição aleatória (100%) alcançam *speedup* de até 2 vezes, os testes com distribuição 80% (maioria *kernels* pequenos) conseguem alcançar *speedup* de até 3 vezes, conforme apresentado na Figura 5.4(b). Isso acontece, pois, quanto maior a quantidade de *kernels* pequenos, mais *kernels* poderão rodar concorrentemente.

Por outro lado, a Figura 5.4(c) mostra que o coeficiente de 20% (maioria *kernels* grandes) alcança *speedup* máximo de apenas 1.6. Quando a maioria dos *kernels* são grandes, a tendencia é que o nível de paralelismo seja reduzido.

Tempo de Resposta

O impacto na média dos tempos de resposta também foi avaliado para cada uma das dimensões. A Figura 5.5 apresenta o impacto no tempo de resposta quando comparadas as implementações sequenciais com as implementações paralelas com múltiplas filas de execução.



(a) Coeficiente 100%

(b) Coeficiente 80%

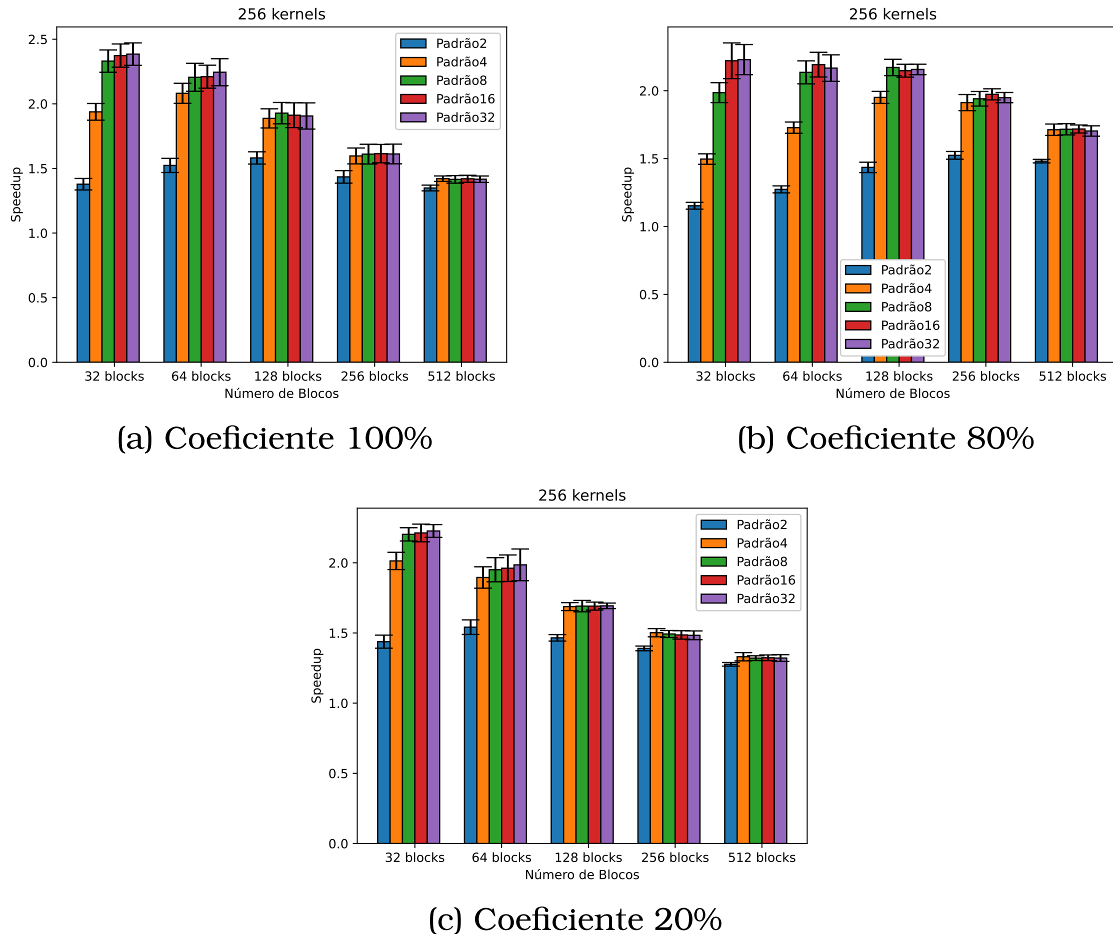
(c) Coeficiente 20%

Figura 5.5: Proporção de melhora da média do tempo de resposta da execução paralela em relação ao tempo de resposta da execução sequencial para 256 *kernels*, usando a estratégia padrão.

Os gráficos foram muito semelhantes ao gráfico do tempo de execução. Essa semelhança acontece porque a ordem dos *kernels* não foi alterada, parâmetro de maior relevância para o tempo médio de resposta. Sendo assim, mesmo que um ou outro kernel executem concorrentemente, o que vai definir a melhora na média do tempo de resposta será o tempo de execução final dos *kernels*.

Tempo de Duração

Nesta seção foi medido o impacto no tempo de duração dos *kernels* quando se tem uma execução paralela, comparada com a execução sequencial. A Figura 5.6 apresenta o impacto medido na execução de 256 *kernels* e coeficientes de distribuição de *kernels* iguais a 100%, 80% e 20%, respectivamente.



(a) Coeficiente 100%

(b) Coeficiente 80%

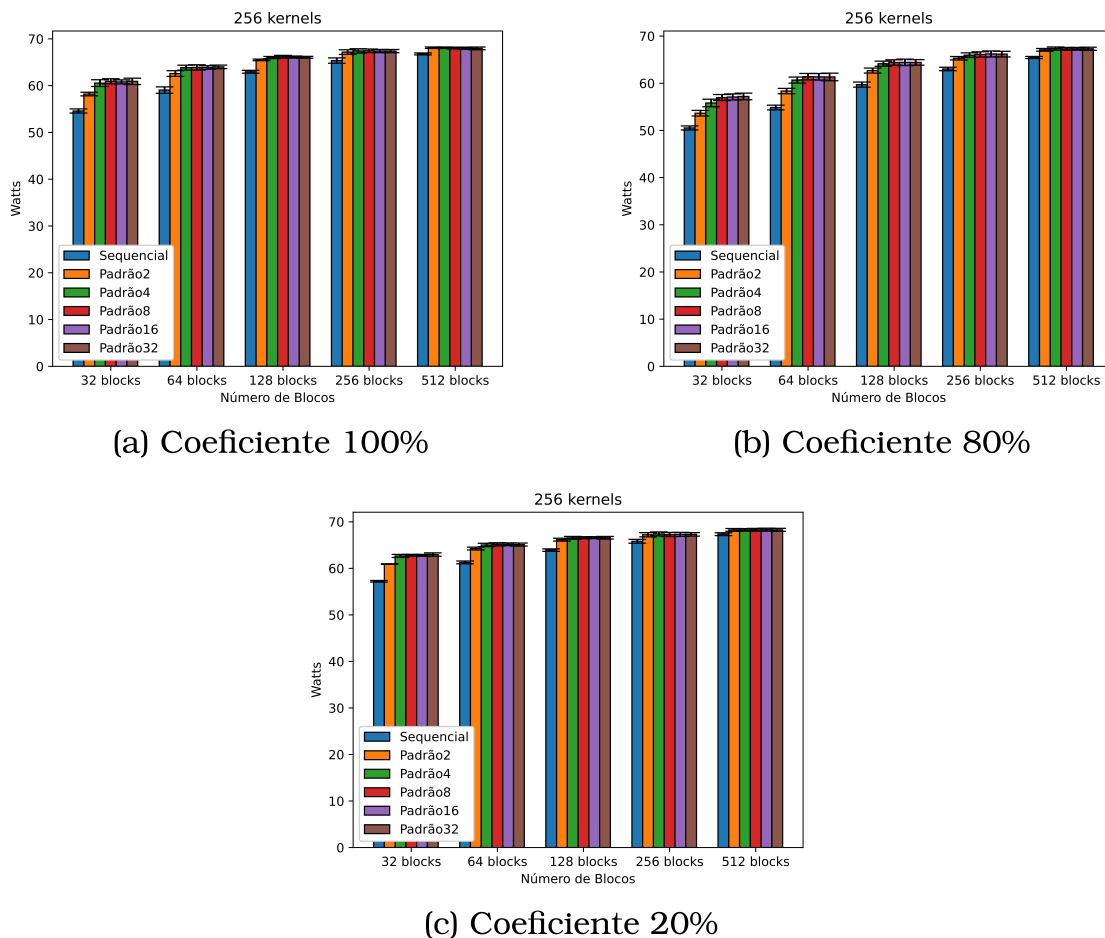
(c) Coeficiente 20%

Figura 5.6: Impacto no tempo de duração da execução paralela em relação à execução sequencial para 256 *kernels*, usando a estratégia padrão.

Os maiores impactos nos tempos de execução dos *kernels* acontecem quando se tem *kernels* menores, pois o nível de concorrência tende a aumentar, já que mais *kernels* cabem juntos na GPU. É natural que a duração de cada kernel aumente, pois o número de multiprocessadores é limitado e, sempre que *kernels* são lançados juntos, o inicio do kernel seguinte se dará nos multiprocessadores que possuem recursos disponíveis. Sendo assim, apesar de os *kernels* estarem iniciando com antecedência ao aproveitar os multiprocessadores livres, eles acabam levando mais tempo para executar, já que não tinham todos os recursos de processamento disponíveis quando iniciaram. Isso também explica os resultados diferentes entre os coeficientes de distribuição dos *kernels*. O coeficiente 80%, maioria de *kernels* menores, gera maior impacto no tempo de processamento que o coeficiente 20%.

Nesta seção, foi mensurado o consumo energético da estratégia ao se alterar o número de filas. Esse consumo foi avaliado em joules e média de watts da placa.

A Figura 5.7 apresenta a média de watts utilizado pela placa para várias configurações de filas de execução. Com o aumento das filas, a média dos watts tende a aumentar, já que tende a maximizar a utilização da GPU. Da mesma forma, *kernels* maiores tendem a manter um nível de utilização da GPU maior que *kernels* menores.



(a) Coeficiente 100%

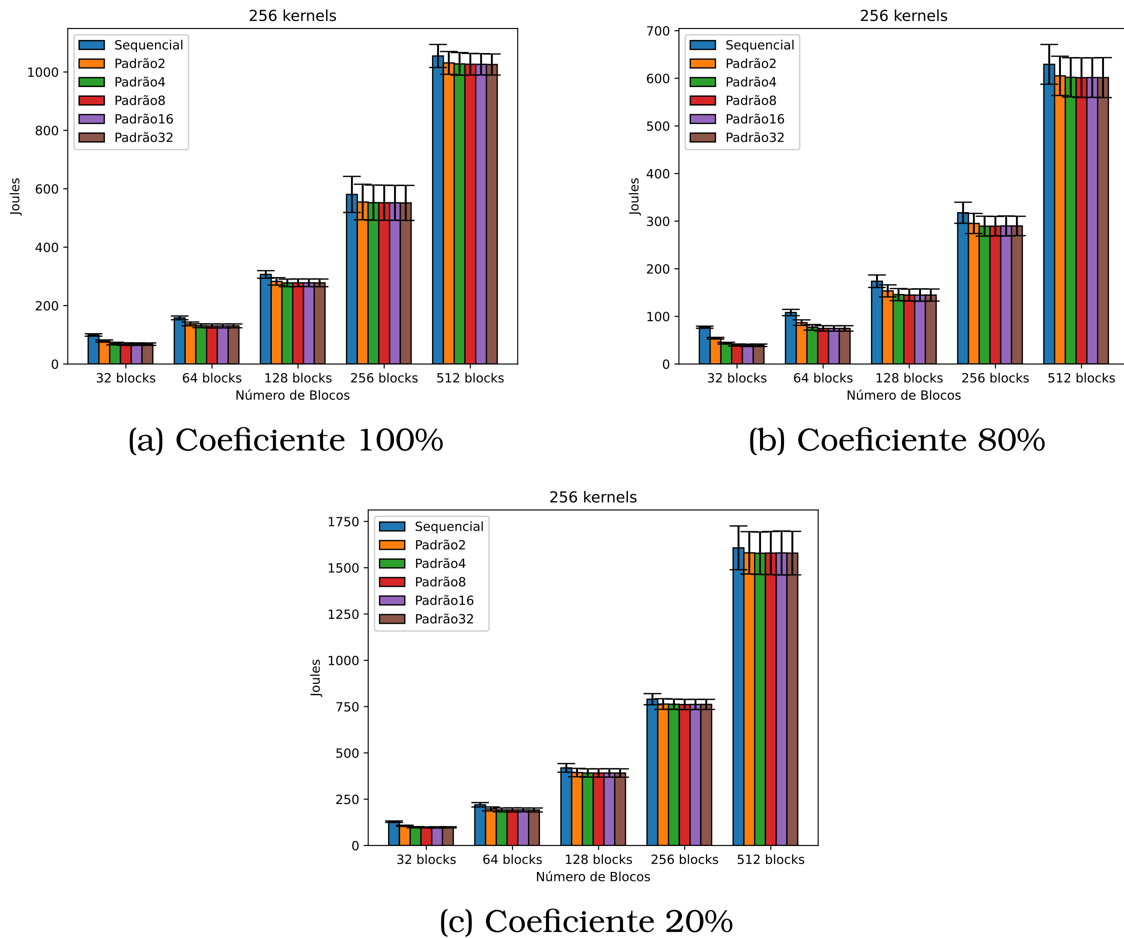
(b) Coeficiente 80%

(c) Coeficiente 20%

Figura 5.7: Impacto na média dos watts da execução paralela em relação à execução sequencial para 256 *kernels*, usando a estratégia padrão.

Por outro lado, a Figura 5.8 apresenta o impacto no consumo (em joules) em cada cenário de acordo com o número de filas de execução. Quando os *kernels* são menores, o aumento no número de filas tem impacto positivo, reduzindo o consumo energético, mas conforme o tamanho dos *kernels* aumenta a diferença de se aumentar o número de filas deixa de fazer diferença. Os joules consumidos variam segundo duas variáveis, os watts e o tempo de execução. Fazendo a associação da quantidade de watts com o tempo de execução em cada dimensão, conclui-se que o fator principal de diferença no

consumo em joules entre as estratégias é o tempo de execução. Sendo assim, quanto maior o tempo de execução, maior será o consumo energético.



(a) Coeficiente 100%

(b) Coeficiente 80%

(c) Coeficiente 20%

Figura 5.8: Impacto da quantidade de joules consumido durante a execução paralela em relação à execução sequencial para 256 *kernels*, usando a estratégia padrão.

Comportamento dos Kernels

A estratégia Padrão (**P**) não faz nenhum tipo de reordenação, ela apenas submete os *kernels* na ordem em que chegam, distribuindo um kernel para cada fila de execução da GPU. Dessa forma, o paralelismo acontece por acaso, quando coincide de *kernels* que cabem juntos chegarem em sequência. Esse comportamento é apresentado na Figura 5.9, em que a linha do tempo é o eixo x e no eixo y aparecem os kernels sendo executados juntos.

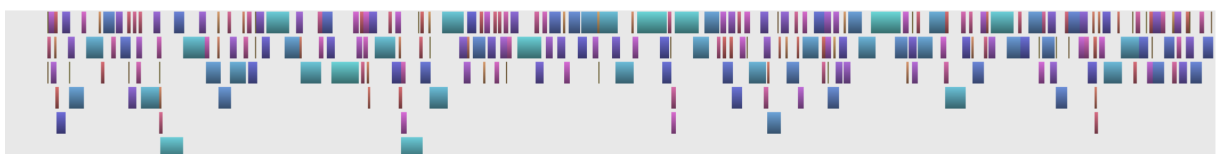


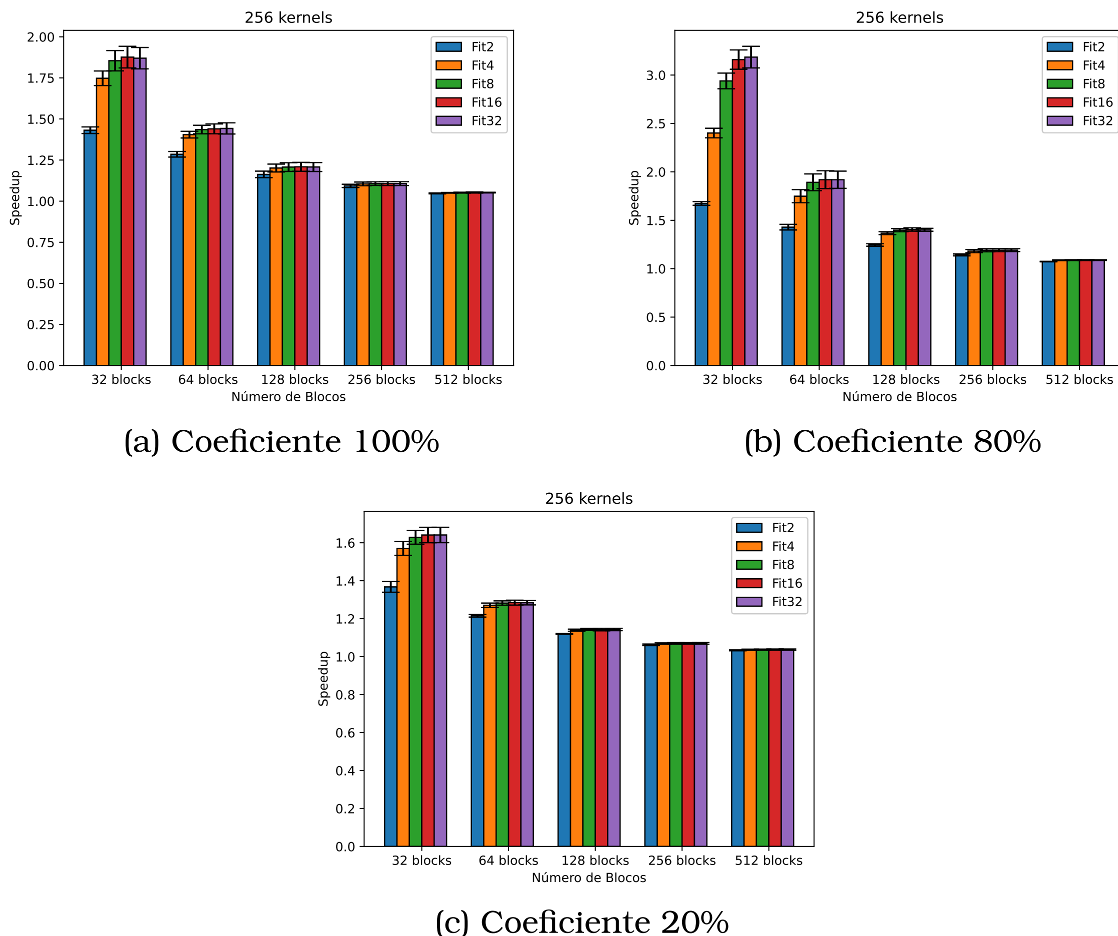
Figura 5.9: Execução paralela dos *kernels* usando a estratégia Padrão (**P**).

5.6.2 Análise da Estratégia Fit (**F**)

Nesta seção, é avaliado o impacto da alteração do número de filas da GPU nos tempos de execução da estratégia Fit, que faz a seleção dos *kernels* na ordem em que chegam sempre que ele caiba para execução com os *kernels* já selecionados antes dele.

Tempo de Execução

A estratégia Fit (**F**) faz a verificação dos *kernels* em ordem, tentando à medida que vão cabendo na GPU, mas ainda assim, os *speedups* dos tempos de execução tiveram resultados muito semelhantes aos observados na estratégia Padrão (**P**), conforme apresentado nas Figuras 5.10.



(a) Coeficiente 100%

(b) Coeficiente 80%

(c) Coeficiente 20%

Figura 5.10: *Speedup* do tempo de execução paralelo em relação ao tempo de execução sequencial para 256 *kernels*, usando a estratégia fit.

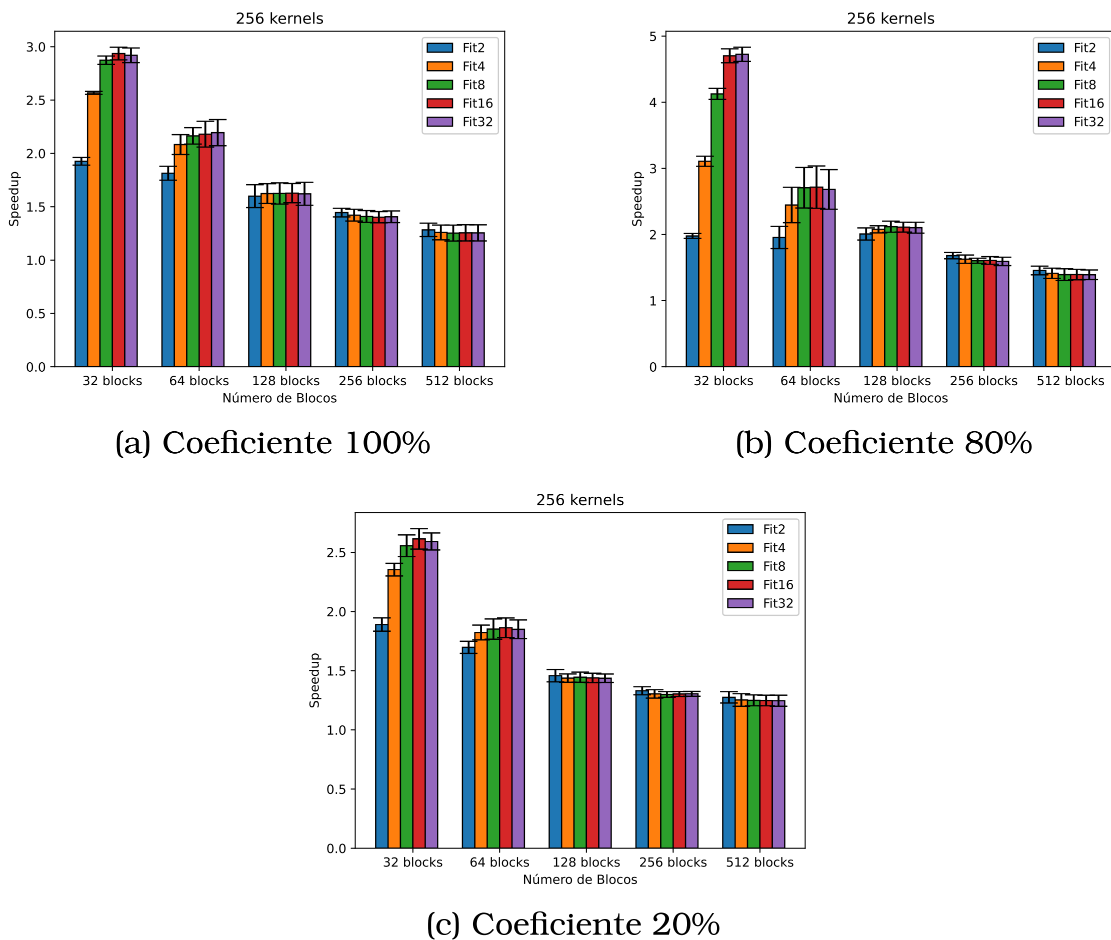
O impacto do número de filas no tempo de execução dos *kernels* depende do tamanho médio deles. Quando o número de blocos dos *kernels* é pequeno, o número de filas tem papel fundamental no tempo de execução. Porém, à medida que os *kernels* ficam maiores, o impacto no tempo de execução vai diminuindo, até o ponto em que não faz diferença se está rodando com 32

filas ou apenas 2, como é o caso de uma execução com 256 *kernels* cujos blocos estão limitado em 512.

Tempo de Resposta

A média do tempo de resposta dos *kernels* é otimizada quando o número de filas de execução aumenta. Porém, *kernels* grandes contendo mais de 128 blocos, a tendência é que 2 filas tenham o mesmo nível de otimização que 32 filas, conforme apresentado na Figura 5.11.

As Figuras 5.11(b) e 5.11(c) apresentam a proporção de melhoria da média dos tempos de resposta para um coeficiente de 80% e 20%, respectivamente. Elas mostram que *kernels* maiores tendem a reduzir o nível de otimização gerado pelo número de filas, já que menos *kernels* poderão ser executados em conjunto.



(a) Coeficiente 100%

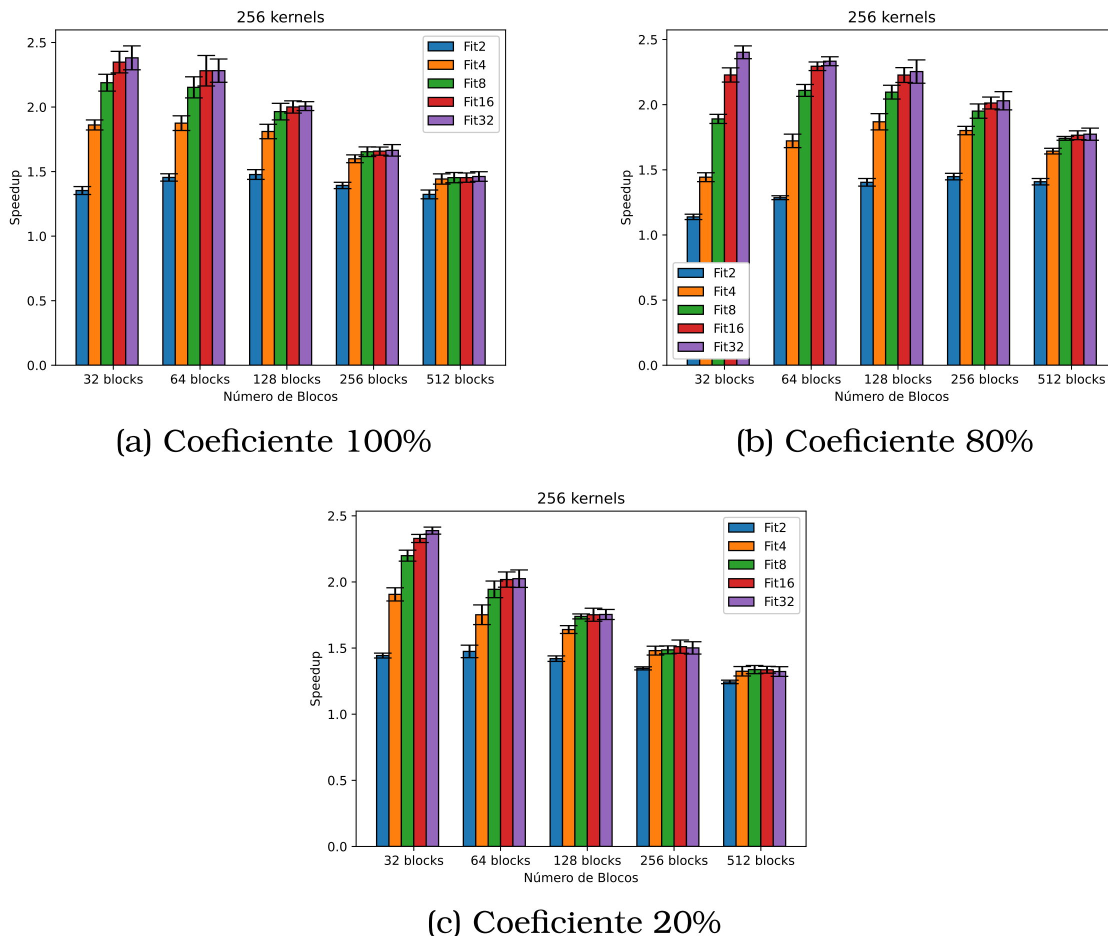
(b) Coeficiente 80%

(c) Coeficiente 20%

Figura 5.11: Proporção de melhora da média do tempo de resposta da execução paralela em relação ao tempo de resposta da execução sequencial para 256 *kernels*, usando a estratégia fit.

Tempo de Duração

O tempo de duração dos *kernels* para a estratégia **F** possui resultado semelhante à estratégia padrão (**P**), quanto maior o nível de paralelismo dos *kernels* maior é o impacto no tempo de duração deles. Dessa forma, *kernels* maiores, por não permitirem alto nível de concorrência, já que sozinhos ocupam grande parte da GPU, tendem a ter um impacto menor. Esse comportamento é apresentado na Figura 5.12, para os coeficientes 100%, 80% e 20%, respectivamente.



(a) Coeficiente 100%

(b) Coeficiente 80%

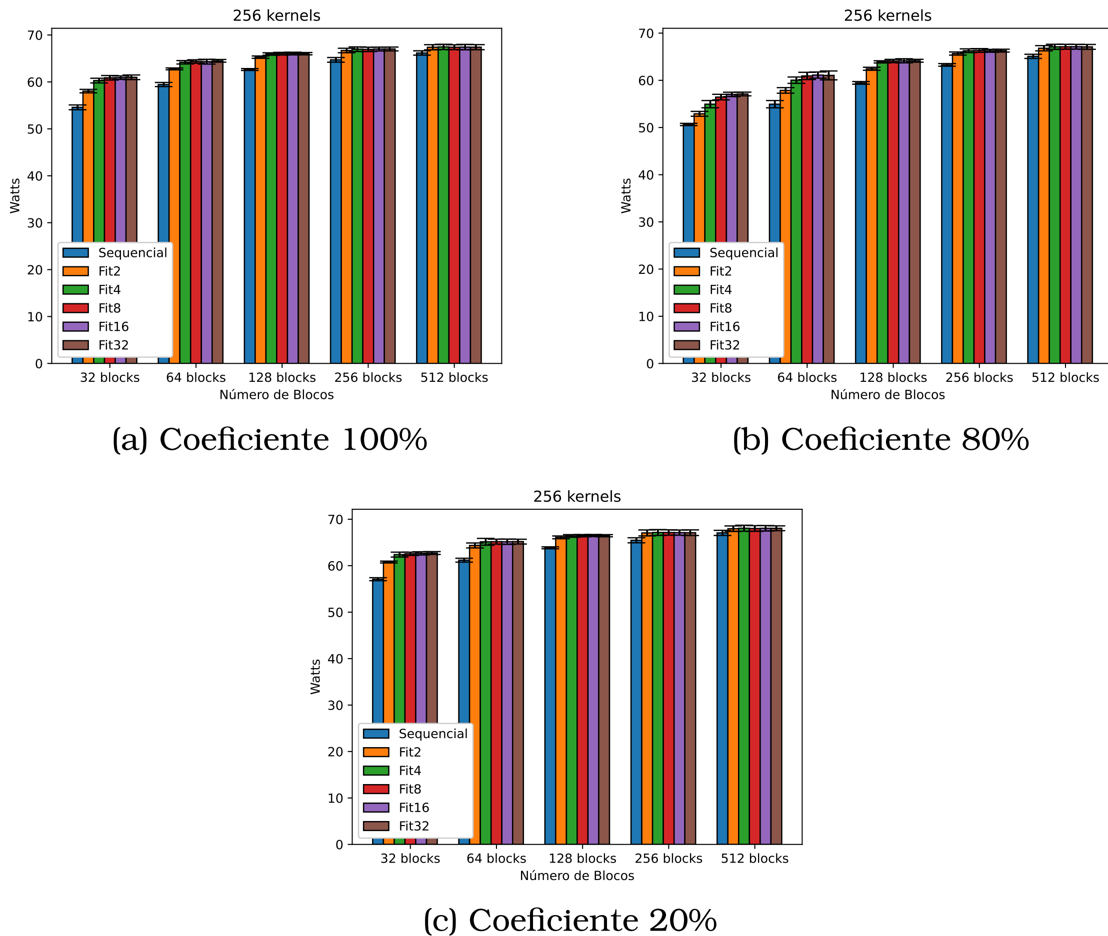
(c) Coeficiente 20%

Figura 5.12: Impacto no tempo de duração da execução paralela em relação à execução sequencial para 256 *kernels*, usando a estratégia fit.

Consumo Energético

O comportamento da média dos watts da estratégia fit (**F**) é apresentado pela Figura 5.13. O gráfico mostra que a média dos watts muda um pouco de um coeficiente para outro, mas a tendência é que o aumento do número de filas implique em uma média um pouco maior. Isso está relacionado com o nível de paralelismo entre os *kernels*, quanto maior a concorrência, maior será a média dos watts. Quando o número de filas aumenta, sem aumentar

a média desse consumo, significa que o nível de utilização da GPU não tenha aumentado, ou seja, o nível de paralelismo não mudou.



(a) Coeficiente 100%

(b) Coeficiente 80%

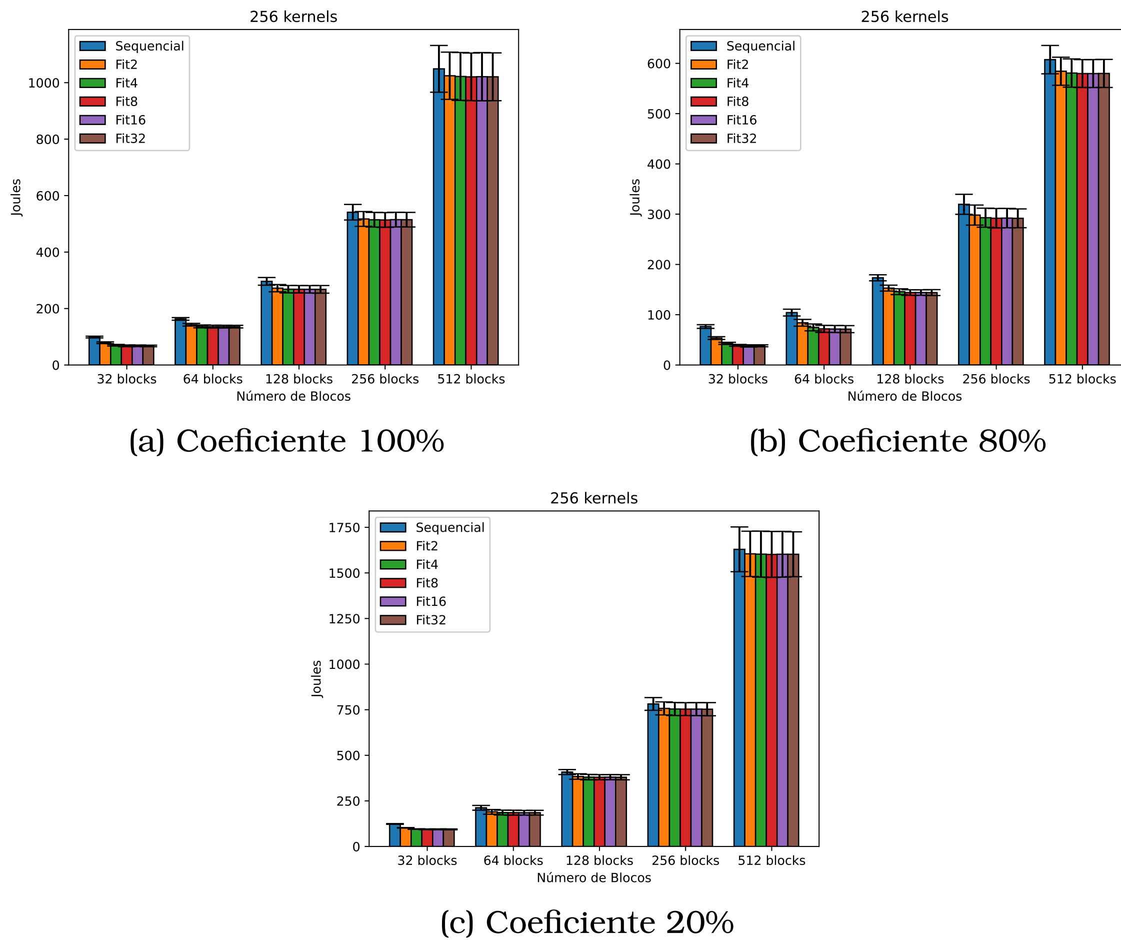
(c) Coeficiente 20%

Figura 5.13: Impacto na média do watts da execução paralela em relação à execução sequencial para 256 *kernels*, usando a estratégia fit.

Apesar de a média dos watts ser um pouco maior nos casos em que o número de filas aumenta, o consumo em joules diminui com esse aumento. Da mesma maneira que na estratégia padrão, esse consumo diminui pois seu cálculo leva em conta os watts, que tem pouca diferença entre as filas, e o tempo de execução, que é menor e acaba sendo o fator determinante dessa métrica. A Figura 5.14 apresenta os consumos em joules da estratégia fit (**F**) para as diversas filas de execução e 256 *kernels*.

Comportamento dos Kernels

A estratégia Fit (**F**), por mais que faça a verificação dos *kernels* em ordem, ela tenta encaixar os *kernels* à medida que vão cabendo na GPU, dessa forma, a ordem de execução pode ser alterada. A Figura 5.15 mostra que essa estratégia consegue melhorar o paralelismo entre os *kernels*, principalmente, no inicio da execução do mesmo. Porém, à medida que se aproxima do final,



(a) Coeficiente 100%

(b) Coeficiente 80%

(c) Coeficiente 20%

Figura 5.14: Impacto na quantidade de joules consumido durante a execução paralela em relação à execução sequencial para 256 *kernels*, usando a estratégia fit.

o nível de paralelismo é reduzido. Isso acontece porque quanto menos *kernels* para executar, menor serão as chances de encontrar *kernels* que caibam juntos na GPU. Além disso, quanto maior a quantidade de *kernels* sendo executados juntos, maior será o impacto no tempo de duração de cada um deles. Da mesma maneira, menor será a precisão na estimativa dos tempos de inicio e fim, interferindo na ordem de finalização dos *kernels*, e consequentemente na alocação dos *kernels* remanescentes nas filas apropriadas.

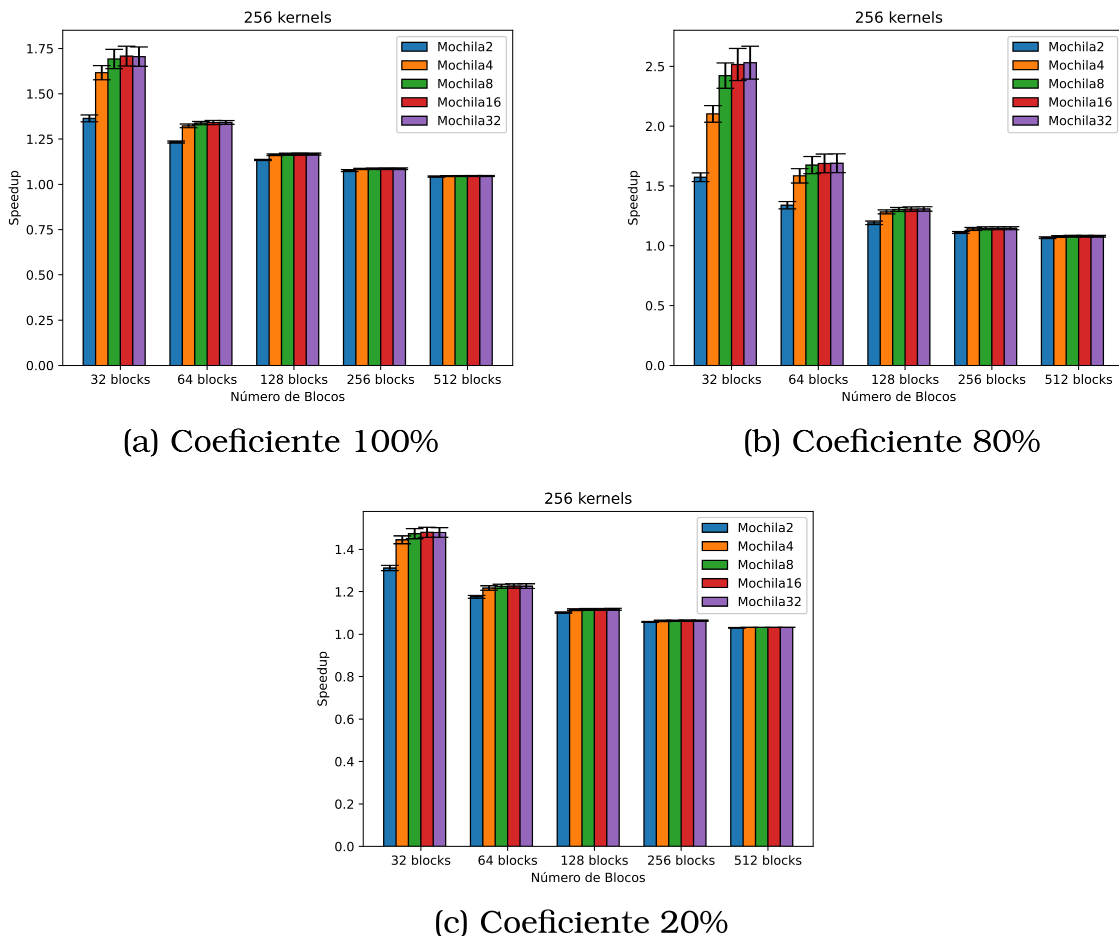
Figura 5.15: Execução paralela dos *kernels* usando a estratégia fit (**F**).

5.6.3 *Análise da Estratégia da Mochila Unidimensional (**ML**)*

Os resultados apresentados nesta seção mostram o comportamento da estratégia da mochila, conforme se aumenta o número de filas da GPU.

Tempo de Execução

A estratégia da mochila (**ML**) pode ser otimizada com o aumento do número de filas de execução. Porém, assim como as estratégias discutidas anteriormente, essa melhora é limitada pelo tamanho dos *kernels*. Quanto menores os *kernels*, maior será a vantagem do aumento no número de filas, e quanto maiores, menor é essa vantagem, sendo que em alguns casos esse aumento será indiferente, como mostra a Figura 5.16.



(a) Coeficiente 100%

(b) Coeficiente 80%

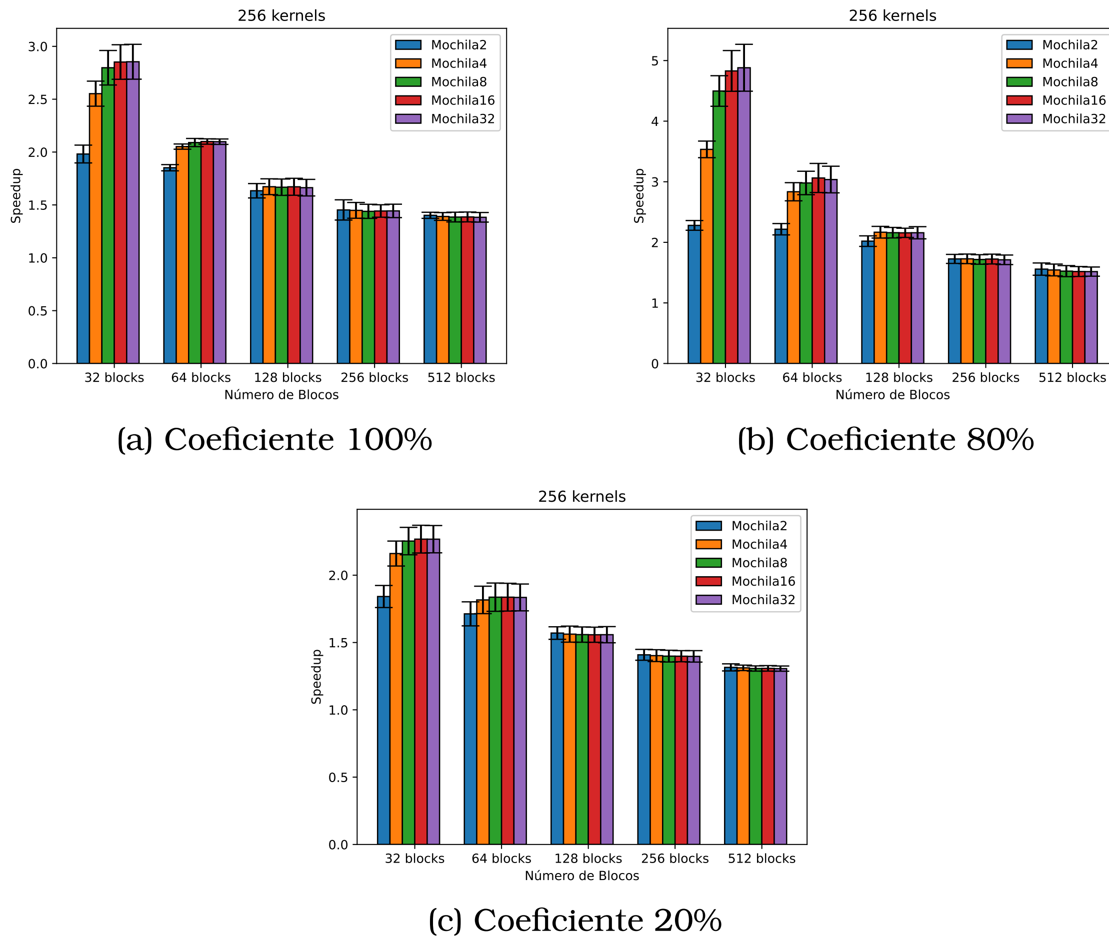
(c) Coeficiente 20%

Figura 5.16: *Speedup* do tempo de execução paralelo em relação ao tempo de execução sequencial para 256 *kernels*, usando a estratégia da mochila.

Tempo de Resposta

Mesmo para *kernels* grandes, o tempo de resposta pode ser 1,5 vezes melhor ao se utilizar 2 filas de execução. A Figura 5.17 mostra que *kernels* menores podem otimizar esse valor ainda mais, já que conseguem executar

mais *kernels* concorrentemente. Quando *kernels* maiores estão sendo executados, ficam limitados pelos seus tamanhos, não havendo diferença entre as configurações de filas de execução.



(a) Coeficiente 100%

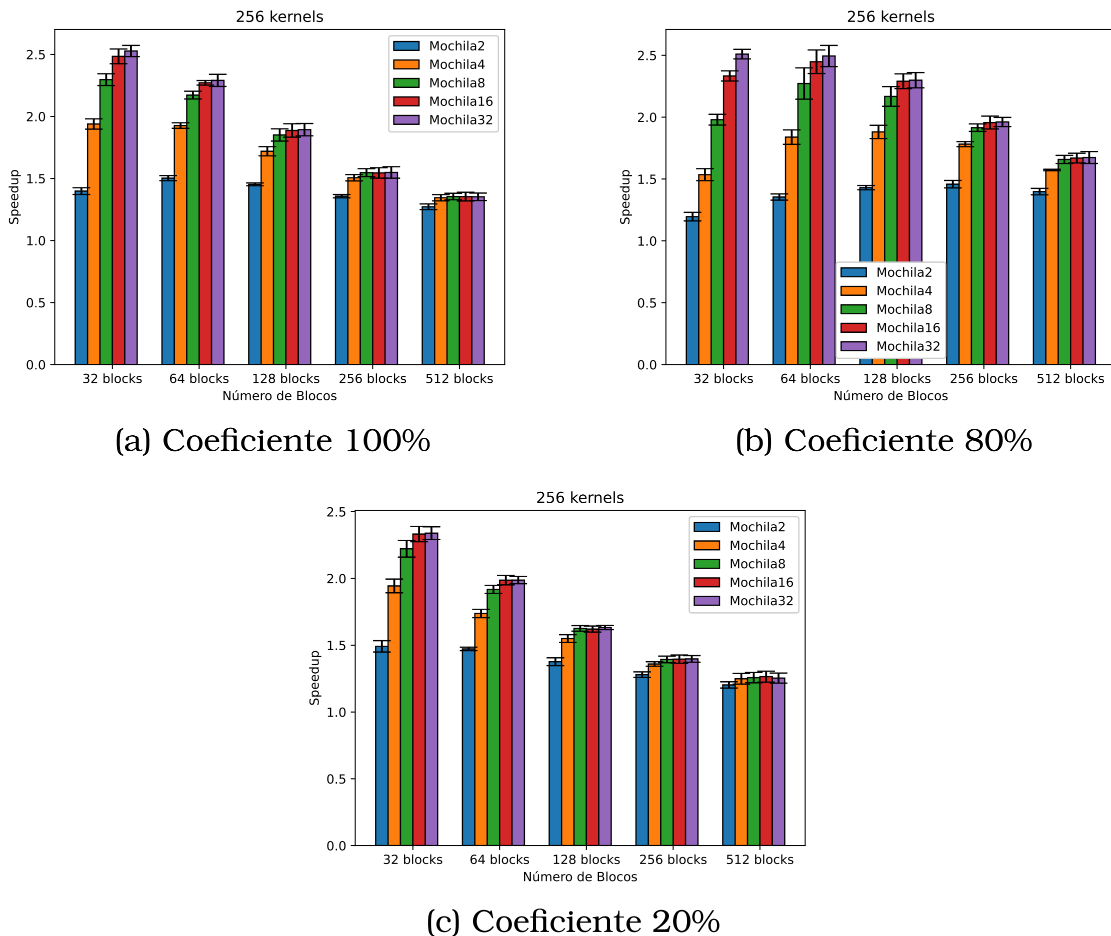
(b) Coeficiente 80%

(c) Coeficiente 20%

Figura 5.17: Proporção de melhora da média do tempo de resposta da execução paralela em relação ao tempo de resposta da execução sequencial para 256 *kernels*, usando a estratégia da mochila.

Tempo de Duração

O tempo de duração individual de cada kernel aumenta com o aumento no número de filas de execução, mas, por outro lado, o tempo de execução total do sistema tende a diminuir, conforme apresentado pela Figura 5.18. O que acontece é que o nível de paralelismo usando essa estratégia aumenta, e faz com que a concorrência entre os *kernels* impacte no tempo individual de cada um deles. Mas esse impacto vai no sentido inverso do tempo de execução, pois permite que mais *kernels* sejam executados quando recursos de GPU estão disponíveis.



(a) Coeficiente 100%

(b) Coeficiente 80%

(c) Coeficiente 20%

Figura 5.18: Impacto no tempo de duração da execução paralela em relação à execução sequencial para 256 *kernels*, usando a estratégia da mochila.

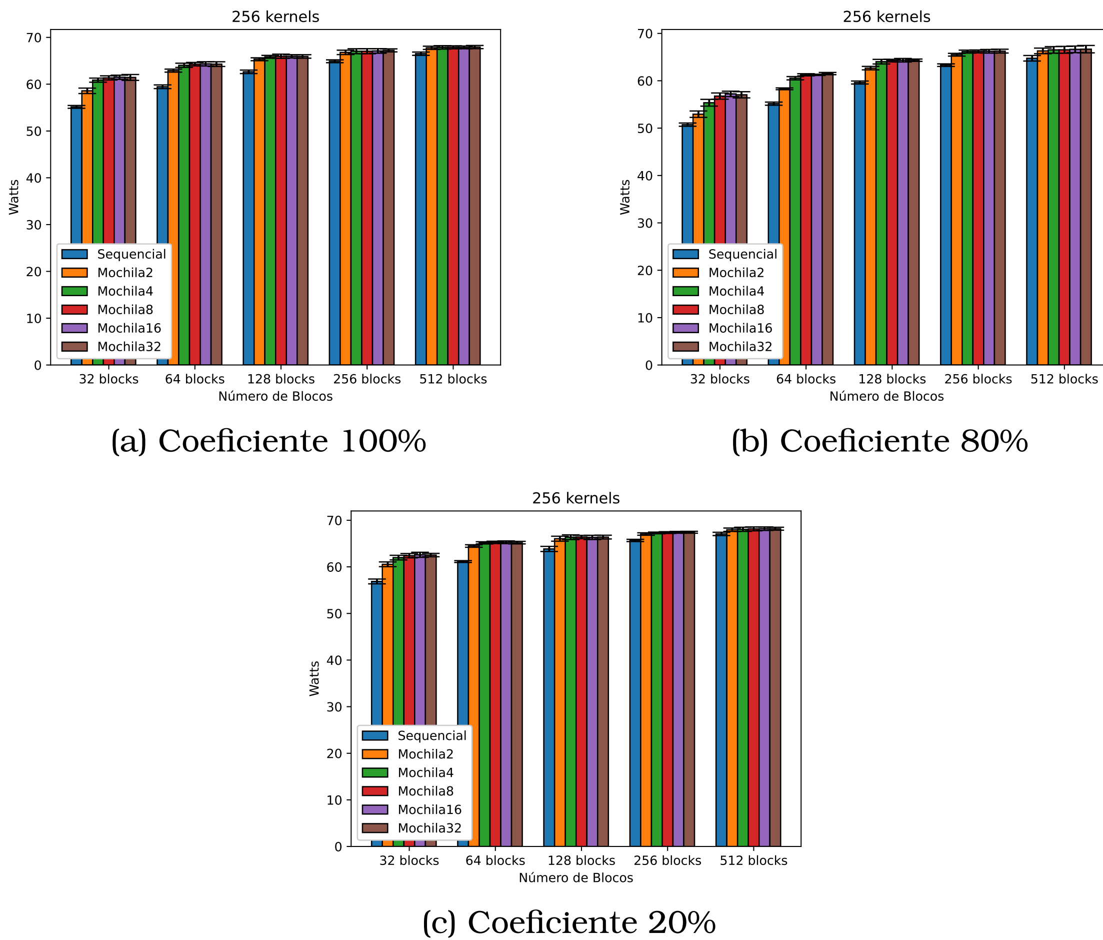
Consumo Energético

Conforme apresentado na Figura 5.19 a média dos watts tem um pequeno acréscimo quando se aumenta o número de filas de execução, porém o consumo em joules diminuiu com o aumento do número de filas. Da mesma maneira que as estratégias anteriores, isto acontece porque o consumo em joules é proporcional ao tempo de execução, que foi reduzido pelo aumento no número de filas, conforme apresentado na Figura 5.20.

Comportamento dos Kernels

A estratégia da mochila (**ML**) é a única estratégia que faz a verificação de várias ordens de execução. Dessa forma, espera-se um nível de paralelismo maior ao executar os *kernels* usando essa estratégia. A Figura 5.21 apresenta o comportamento dos *kernels* ao fazer a reordenação usando essa estratégia.

Assim como na estratégia fit (**F**), o nível de paralelismo dos *kernels* vai reduzindo à medida que os *kernels* vão finalizando. Isso acontece pelo mesmo motivo da estratégia anterior, a precisão da estimativa do tempo dos *kernels*



(a) Coeficiente 100%

(b) Coeficiente 80%

(c) Coeficiente 20%

Figura 5.19: Impacto na média dos watts da execução paralela em relação à execução sequencial para 256 *kernels*, usando a estratégia fit.

vai se reduzindo, bem como o número de *kernels* disponíveis para serem selecionados para execução paralela é menor.

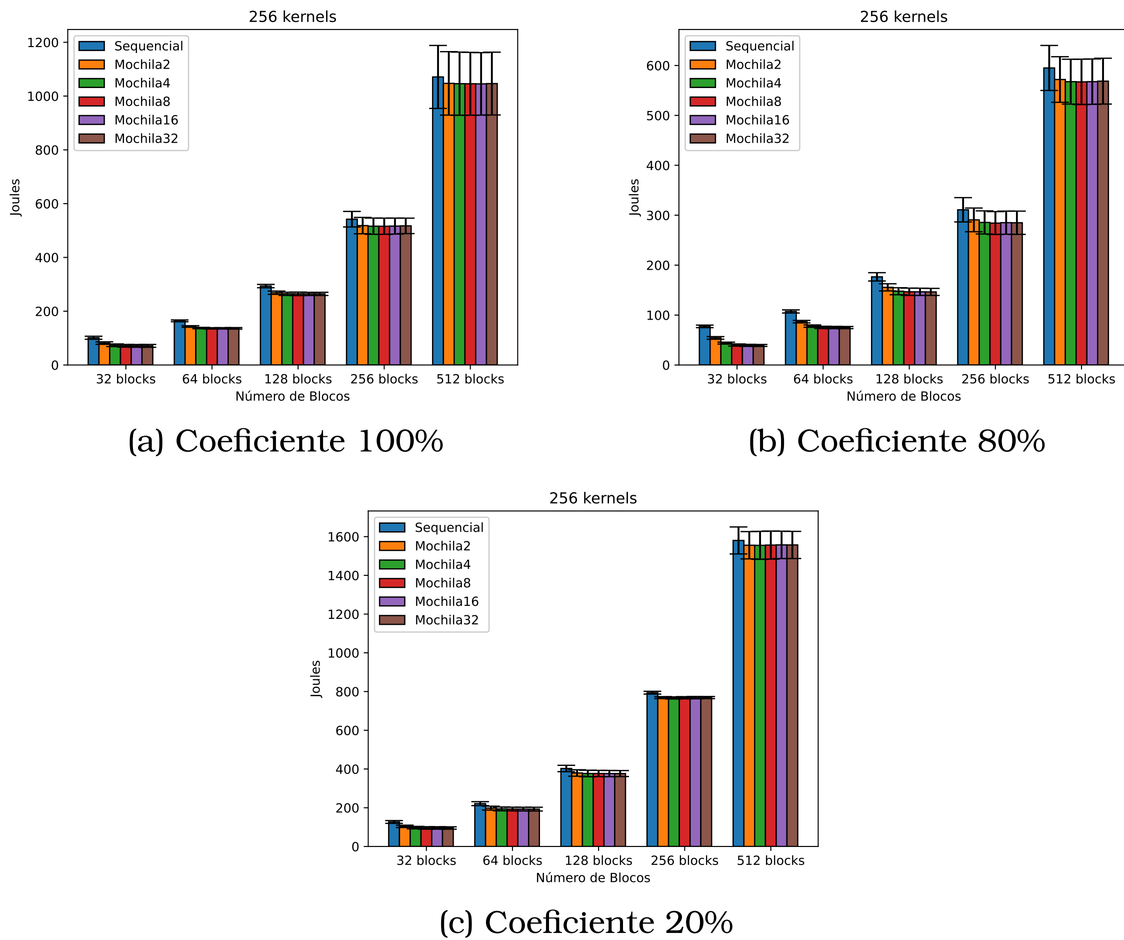
5.7 Comparação entre as Estratégias

Nesta seção, as estratégias foram comparadas para identificar quais métricas podem ser otimizadas por cada uma delas. Para simplificar a comparação entre elas, as filas de execução foram configuradas em 32.

A Tabela A.1 apresentada no Apêndice A, apresenta a média dos tempos de execução de cada uma delas e o desvio padrão para cada dimensão do problema. Esses resultados são discutidos a seguir.

Tempo de Execução

A Figura 5.22 apresenta os *speedups* dos tempos de execução das estratégias fit (**F**) e da mochila (**ML**), quando comparadas com a estratégia padrão (**P**). Elas mostram que independente do coeficiente de distribuição dos *kernels*, maiores ou menores, o tempo de execução entre as estratégias permanece pra-



(a) Coeficiente 100%

(b) Coeficiente 80%

(c) Coeficiente 20%

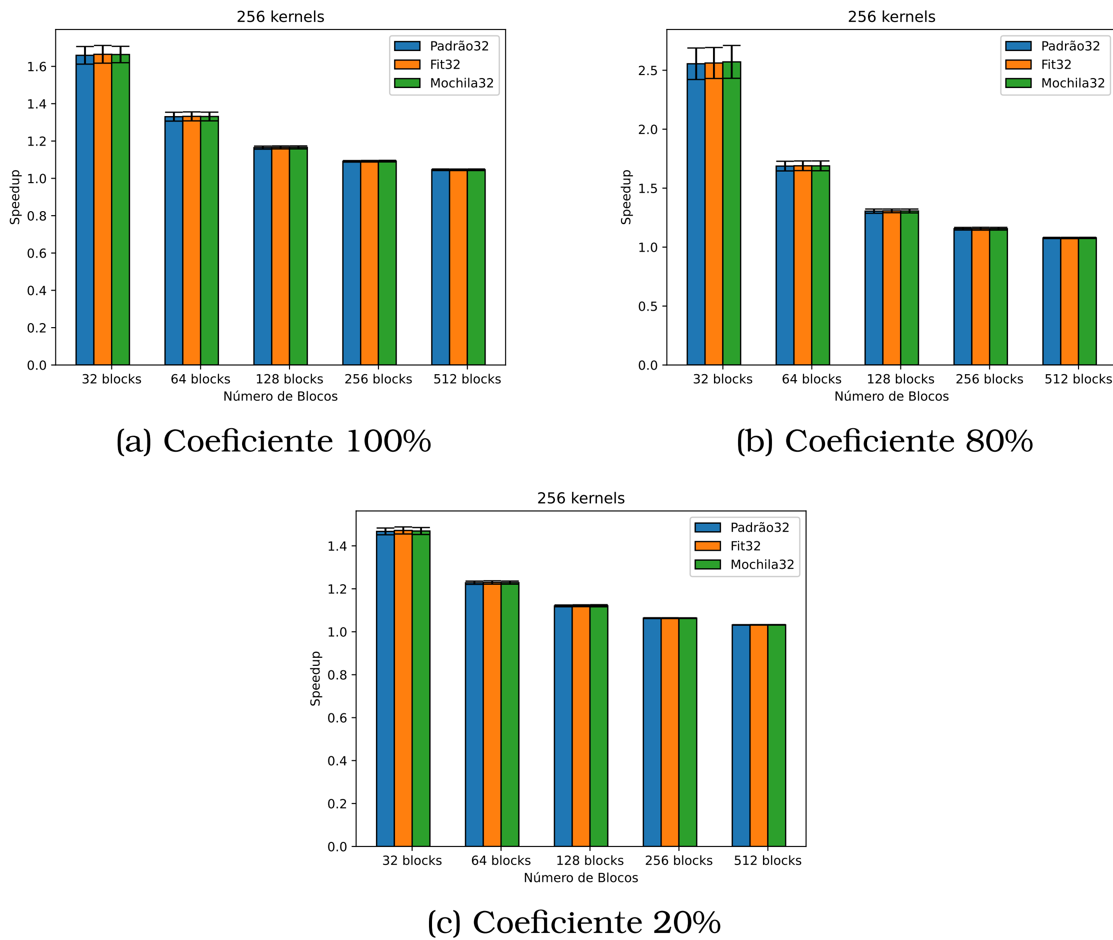
Figura 5.20: Impacto na quantidade de joules consumido durante a execução paralela em relação à execução sequencial para 256 *kernels*, usando a estratégia da mochila.

Figura 5.21: Execução paralela dos *kernels* usando a estratégia da Mochila (**ML**).

ticamente o mesmo. Ou seja, ainda que a reordenação de *kernels* altere o nível de paralelismo entre eles, o tempo de execução médio não se altera.

Tempo de Resposta

Apesar do tempo de execução não se alterar entre as estratégias, o tempo de resposta é afetado por elas, já que a ordem de execução é alterada. A Figura 5.23 mostra que a estratégia que melhor reordena os *kernels* para fins de tempo de resposta é a estratégia da mochila (**ML**). Na sequência, a estratégia



(a) Coeficiente 100%

(b) Coeficiente 80%

(c) Coeficiente 20%

Figura 5.22: *Speedup* do tempo de execução paralelo em relação ao tempo de execução sequencial para 256 *kernels*.

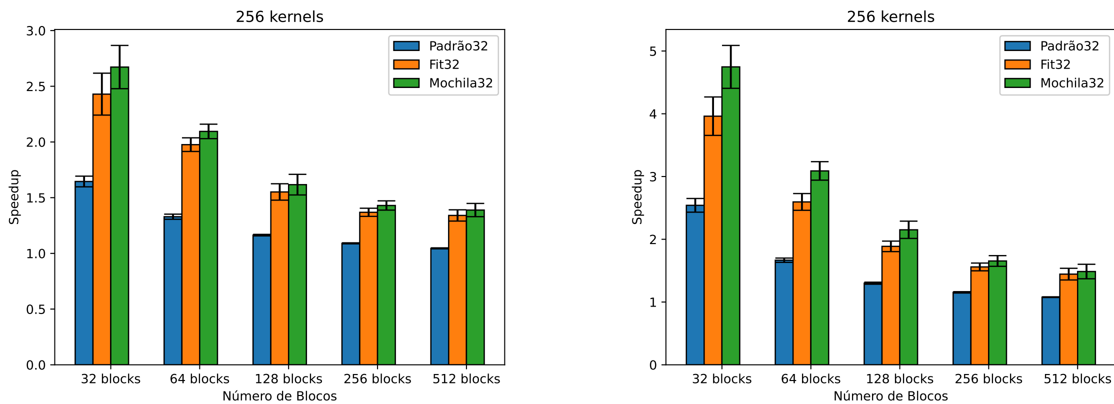
fit (**F**) apresenta o melhor resultado, finalizando com a estratégia padrão (**P**) que simplesmente executa os *kernels* na ordem em que chegam.

Tempo de Duração

O impacto no tempo de duração dos *kernels* também é muito semelhante entre as estratégias, conforme apresentado na Figura 5.24. Dessa forma, o nível de paralelismo tende a ser muito semelhante também, já que é o principal fator que determina o impacto no tempo de duração.

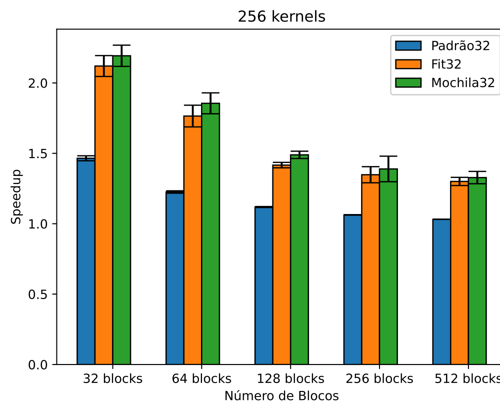
Consumo Energético

A Figura 5.25 mostra que, independente do tamanho dos *kernels*, as estratégias apresentam a média dos watts muito parecido. Dessa forma, o que vai determinar o consumo em joules é o tempo de execução. Porém, como já apresentado anteriormente, o tempo de execução entre as estratégias também é muito parecido. Logo, o consumo energético final não vai ser alterado pela estratégia escolhida, como apresenta a Figura 5.26.



(a) Coeficiente 100%

(b) Coeficiente 80%



(c) Coeficiente 20%

Figura 5.23: Proporção de melhora da média do tempo de resposta da execução paralela em relação ao tempo de resposta da execução sequencial para 256 *kernels*.

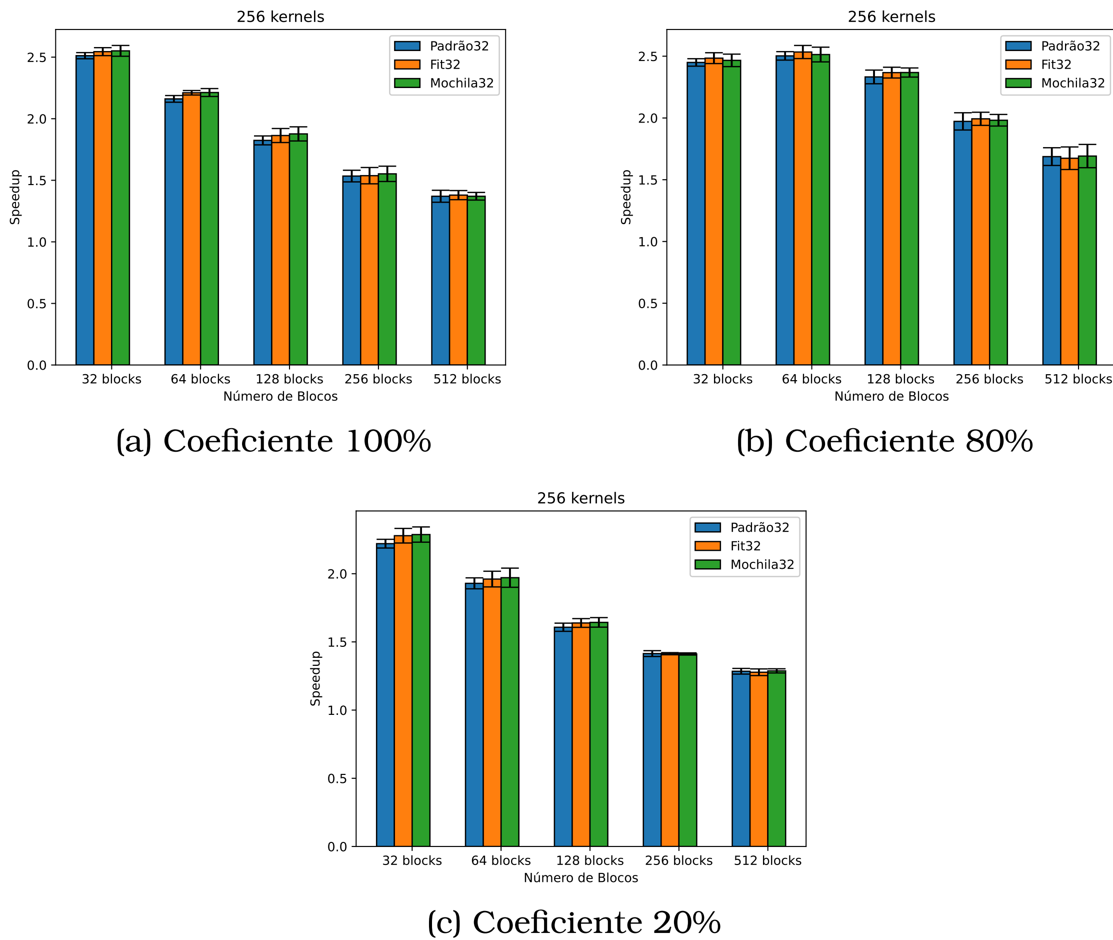
5.8 Análise das Filas de Execução

Nesta seção foi realizada uma análise mais detalhada das filas de execução para entender o comportamento dos *kernels* ao se alterar a ordem de execução.

Execução Concorrente

A GPU permite a execução concorrente entre os *kernels* sempre que existem recursos disponíveis. Os recursos que devem estar disponíveis são: memória compartilhada, número de registradores e número de threads. O número de threads que pode estar ativa na GPU ao mesmo tempo é limitada pelo número de *warps* ativos por SM. Na placa testada neste capítulo o número de *warps* que podem estar ativos por SM é de 64. Como são 13 SMs na placa, o número de *warps* máximo é de 832 (ou 26624 threads, desde que hajam *warps* suficientes para elas).

A Figura 5.27 apresenta a execução de um conjunto de *kernels* em queo



(a) Coeficiente 100%

(b) Coeficiente 80%

(c) Coeficiente 20%

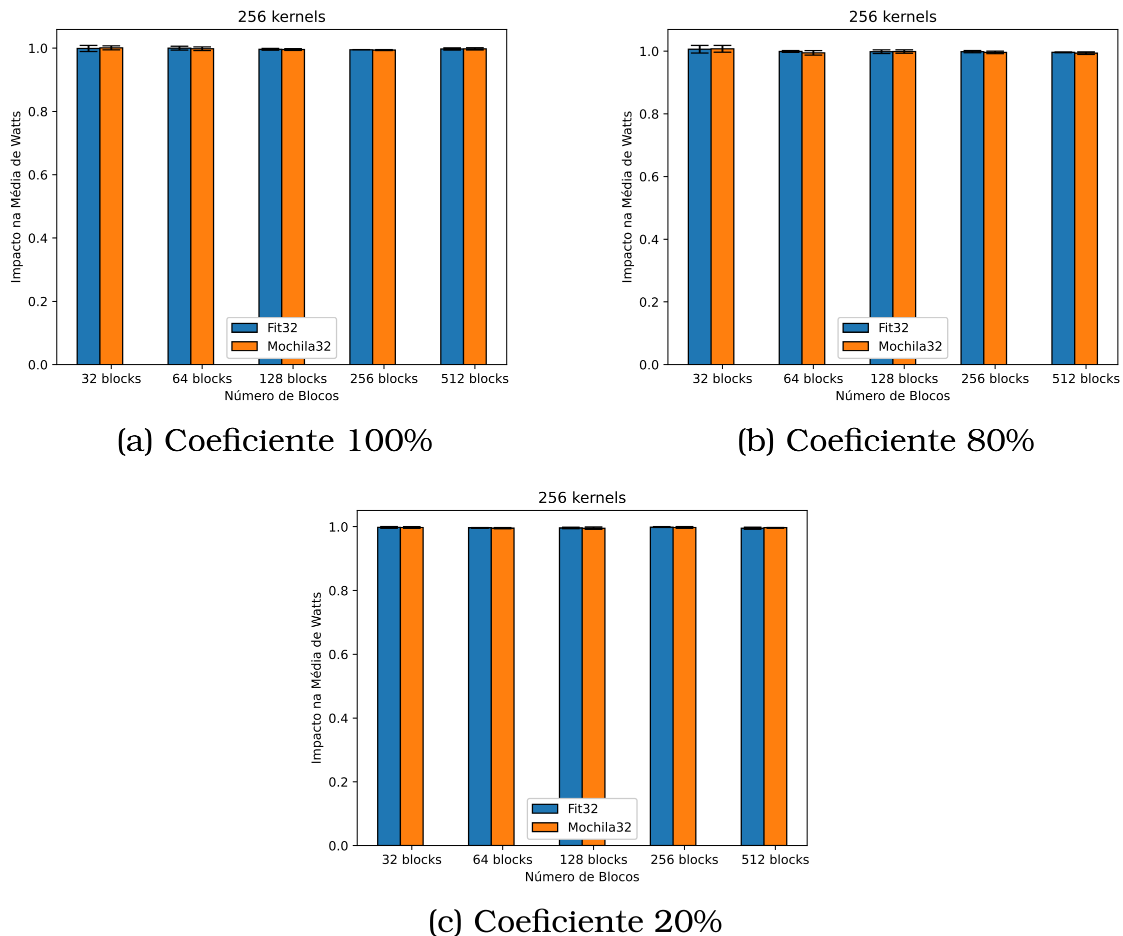
Figura 5.24: Impacto no tempo de duração da execução paralela em relação à execução sequencial para 256 *kernels*.

primeiro está configurado com 104 blocos e 256 threads, o que resulta em um total de $104 \times 256 = 26624$ threads, que é exatamente o máximo de threads da GPU. Dessa forma, nenhum outro kernel pode executar junto com ele nesse cenário.

Na Figura 5.28 é apresentado o que aconteceria se no lugar de 104 blocos o kernel tivesse apenas 103 blocos. Como o número de blocos foi reduzido, 256 threads estarão livres e este número é exatamente o número de threads de cada bloco do kernel 02. Sendo assim, um bloco do kernel 02 poderá iniciar sua execução, mesmo que os outros blocos ainda estejam aguardando a finalização do kernel 01. Isso explica também o motivo de na primeira execução, quando o kernel 02 executa sozinho, ele levar um tempo muito menor para executar que na execução quando inicia com apenas um bloco.

Independência entre os Kernels

Quando os *kernels* são escalonados em uma mesma fila de execução da GPU, é assumido que há dependência entre eles e não haverá execução concorrente. Mesmo que tenham recursos de GPU disponíveis, uma nova execu-



(a) Coeficiente 100%

(b) Coeficiente 80%

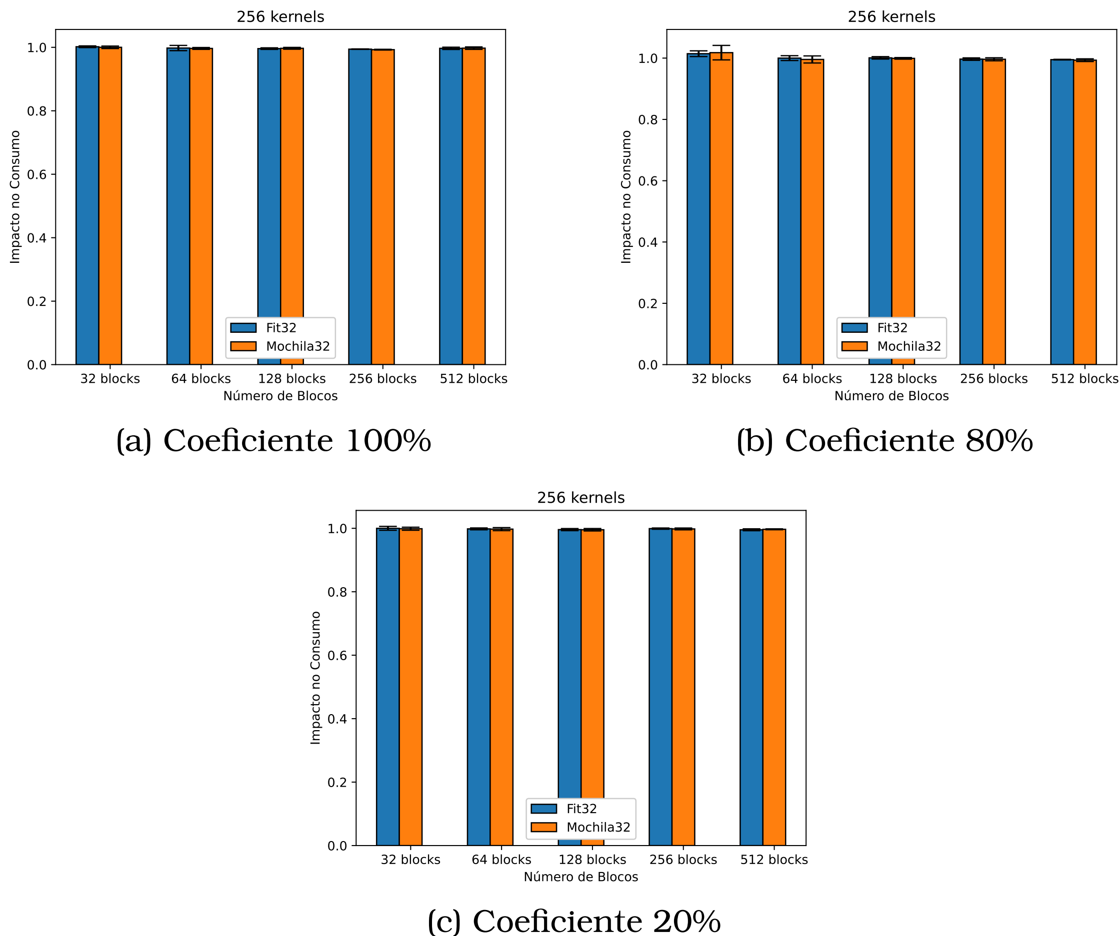
(c) Coeficiente 20%

Figura 5.25: Impacto na média dos watts da execução paralela em relação à execução sequencial para 256 *kernels*.

ção só irá iniciar após a finalização do kernel em execução. Agora, ao distribuir os *kernels* em filas diferentes, está sendo determinado a independência entre eles. Dessa forma, sempre que houver recursos de GPU disponíveis, *kernels* de filas diferentes poderão executar concorrentemente.

A Figura 5.29 apresenta um cenário em que 5 *kernels* foram submetidos em 5 filas diferentes. Como foram submetidos em filas diferentes, serão considerados como independentes pela GPU, podendo ser executados concorrentemente. Porém, isso só vai acontecer caso os cinco caibam juntos na GPU.

Mesmo nos casos em que se tem apenas *kernels* grandes, que não cabem inteiros na GPU, é possível aproveitar algum nível de paralelismo. Quando um kernel inicia sozinho sua execução na GPU, ele utiliza o máximo de recursos disponíveis de acordo com sua configuração de blocos. Isto é, os blocos de um kernel são distribuídos entre os multiprocessadores da GPU (*strem multi-processor*) e conforme esses blocos vão finalizando, os multiprocessadores vão sendo liberados para que o próximo kernel da fila possa iniciar sua execução. Porém, isso só vai acontecer caso os *kernels* sejam indendentes entre si, ou seja, se tiverem sido submetidos em filas diferentes.



(a) Coeficiente 100%

(b) Coeficiente 80%

(c) Coeficiente 20%

Figura 5.26: Impacto no número de joules consumido durante a execução paralela em relação à execução sequencial para 256 *kernels*.

A Figura 5.30 mostra um exemplo de *kernels* que não cabem juntos na GPU sendo executados de forma concorrente. A parte final da execução do primeiro roda concorrentemente com a parte inicial do segundo, e assim sucessivamente. Por isso, é interessante evitar que *kernels* de uma mesma fila sejam executados consecutivamente sem intercalar com ao menos uma outra fila.

Ordem de Execução

A ordem de execução dos *kernels* dependem de dois fatores, o primeiro é a ordem em que foram despachados pelo usuário e o segundo é o tempo de execução dos *kernels* que já estavam na fila antes deles.

O primeiro fator pode ser observado ao despachar cinco *kernels* para execução em cinco filas diferentes. Como foram submetidos em filas diferentes, serão considerados como independentes pela GPU, podendo ser executados concorrentemente. Porém, isso só vai acontecer caso os cinco caibam juntos na GPU. Um problema que poderia haver neste cenário seria o despacho de um kernel grande, que só cabe sozinho na GPU, entre esses *kernels* que ca-

Figura 5.27: O primeiro kernel possui 104 blocos com 256 threads cada um. O limite do número de threads é atingido pelos SMs e nenhum outro kernel pode iniciar até que ele termine.

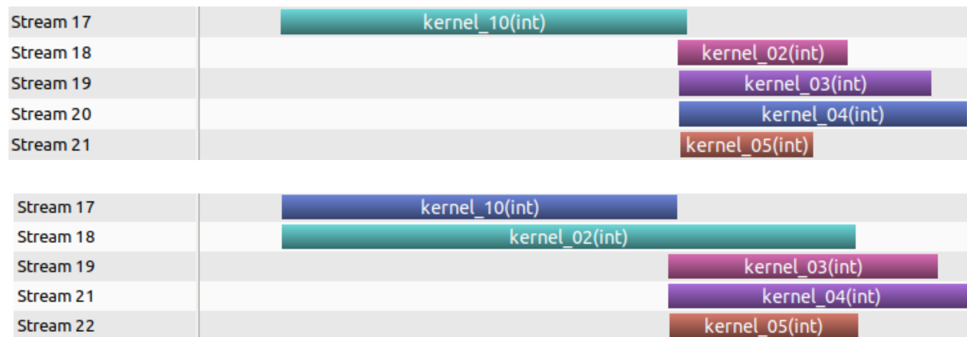


Figura 5.28: O primeiro kernel possui 103 blocos com 256 threads cada um. O limite do número de threads por SM é atingido em 12 SMs, mas um deles ainda tem disponível 256 threads, que é o número de threads do segundo kernel.

bem juntos. Como a GPU não altera a ordem de execução dos *kernels*, eles deverão aguardar que este kernel maior termine para só então iniciar sua execução. Neste caso, haverá uma subutilização da GPU, pois um kernel que não ocupa totalmente a GPU estará executando sozinho. Dessa forma, não basta os *kernels* caberem juntos na GPU, eles devem ser despachados para execução juntos também.

A Figura 5.31 apresenta um exemplo de um grupo de *kernels* em que o primeiro cabe junto com os três últimos, porém, como o segundo kernel é grande, e foi despachado entre eles, a GPU aguardará a finalização do segundo, para só então iniciar a execução dos três últimos.

Apesar de a GPU possuir várias filas de despacho de *kernels*, a fila de execução é única. Conforme o kernel de uma fila termina sua execução, um novo kernel daquela fila é submetido para a fila de execução. Dessa forma, se uma fila possui *kernels* com tempo de execução muito menores que os *kernels* de outra fila, pode acontecer de os *kernels* da fila mais rápida passarem na frente dos *kernels* da fila mais lenta. A Tabela 5.1 apresenta a ordem de execução de um conjunto de *kernels* em que há uma alteração na ordem de execução devido à diferença no tempo de execução dos *kernels* de uma fila. Para simplificar a explicação, foram utilizados apenas três filas de despacho e *kernels* que cabem juntos na GPU. A Figura 5.32 apresenta o resultado dessa execução. Este exemplo mostra que apesar do kernel 05 ter sido despachado pelo usuário depois do kernel 04, ele acaba executando primeiro, pois o kernel 02 finalizou antes do kernel 01, fazendo com que o kernel 05 fosse submetido para a fila de execução primeiro.

Um outro cenário desse comportamento é apresentado na Figura 5.33. Neste exemplo, nem todos os *kernels* cabem juntos na GPU, mas ainda as-

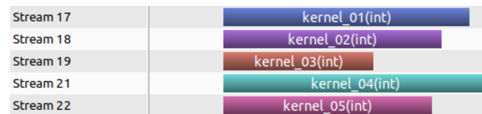


Figura 5.29: Todos os *kernels* cabem juntos na GPU.

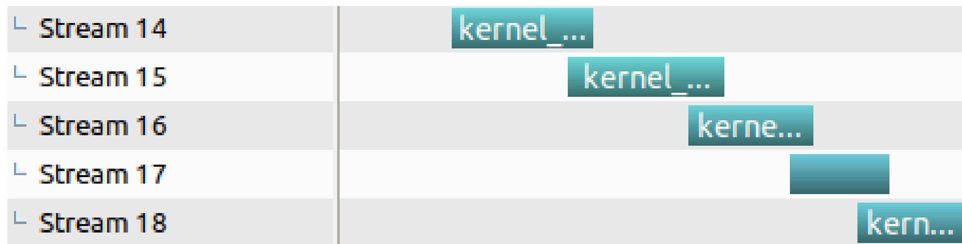


Figura 5.30: Kernels grandes sendo executados em *streams* diferentes.

Tabela 5.1: Ordem de despacho dos *kernels* efetuada pelo usuário. Referente à Figura 5.32

Ordem	Kernel	Fila
1	01	17
2	02	18
3	03	19
4	04	17
5	05	18
6	06	19
7	07	17
8	08	18
9	09	19

sim, existe a troca de ordem de execução. O kernel 11 foi submetido pelo usuário antes do kernel 04, mas executa primeiro por causa do kernel 02, que finaliza primeiro que o kernel 01.

5.9 Ambiente com múltiplas GPUs

O ambiente com múltiplas GPUs possui um multiprocessador Intel(R) Xeon(R) CPU E5-2620 @2.00GHz, 32 GB de RAM e quatro placas de processamento gráfico GeForce GTX TITAN Black. A máquina foi configurada com compilador G++ 7.4.0 e CUDA 10.1. Os *kernels* foram gerados da mesma maneira que o ambiente com uma única GPU.

Como esse ambiente possui quatro GPUs, além da reordenação dos *kernels*, a distribuição dos *kernels* entre as máquinas também deve ser considerado. Dessa forma, as estratégias da seção anterior foram adaptadas para realizar essa distribuição, e seis estratégias foram criadas.

A primeira estratégia de distribuição, chamada de distribuição *padrão*, dis-

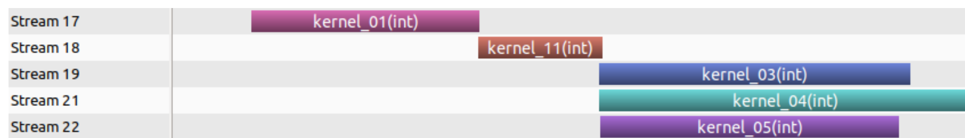


Figura 5.31: Apenas o kernel 11 não cabe na GPU junto com os outros.

Figura 5.32: Um exemplo ideal, quando os *kernels* das três filas de execução cabem juntos na GPU.

tribui igualmente os *kernels* entre as GPUs disponíveis e, posteriormente, os distribui entre as filas de execução das GPUs. Os *kernels* são distribuídos sem nenhuma alteração na ordem de execução, nem análise das cargas de trabalho a serem alocadas em cada máquina.

À segunda estratégia, deu-se o nome de *fit*, pois trata-se de uma adaptação da estratégia *fit* com uma única GPU. Ela itera sobre as GPUs rodando um *fit* de GPU única para cada uma. Dessa forma, ao final da primeira iteração, todas as GPUs recebem os *kernels* iniciais. A partir desse ponto, as GPUs recebem novos *kernels* à medida que seus *kernels* vão finalizando e liberando recursos, usando sempre um *fit* para seleção dos *kernels*.

A terceira estratégia, chamada *mochila*, é semelhante à segunda estratégia com a diferença de executar uma mochila para realizar a seleção de *kernels*. Então ela é uma adaptação do algoritmo da mochila da seção anterior, que itera sobre as GPUs resolvendo o problema da mochila com linearização dos pesos.

Além disso, outras estratégias foram testadas, mas focando na distribuição dos *kernels* entre as máquinas. Essas estratégias utilizam a heurística min-min ou a heurística max-min para fazer a distribuição dos *kernels* entre as GPUs e, depois, fazem a reordenação dos *kernels* usando uma das estratégias: padrão, fit ou mochila. Dessa forma, foram criadas outras seis estratégias, três que utilizam o min-min para fazer a distribuição dos *kernels* e três que utilizam o max-min para fazer essa distribuição.

Tempo de Execução

Os resultados mostraram que independentemente da estratégia de distribuição escolhida, a média dos tempos das máquinas são muito parecidos, conforme apresentado na Figura 5.34. Isso sugere que a forma de escalonamento entre as GPUs não interfere no desempenho final do sistema, assim como observado nas implementações com apenas uma GPU.

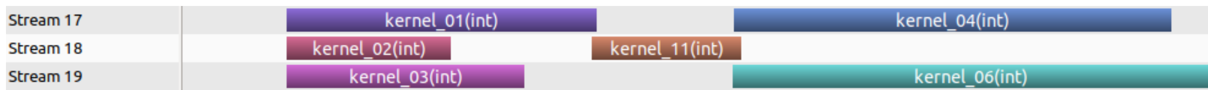


Figura 5.33: Um exemplo da mudança da ordem execução, quando um kernel de uma fila finaliza antes que um kernel que foi despachado primeiro.

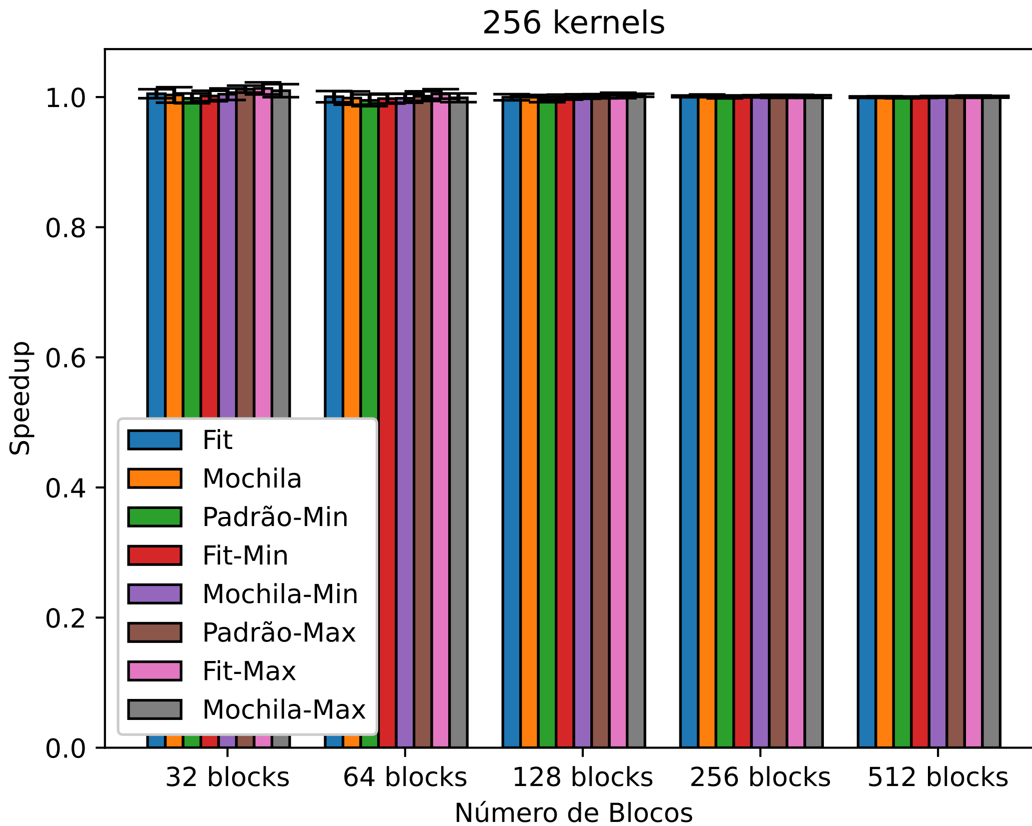


Figura 5.34: *Speedup* das implementações em relação à estratégia padrão, que distribui os *kernels* igualmente entre as GPUs.

Tempo de Conclusão

A Figura 5.35 mostra as médias dos tempos de conclusão dos *kernels* para cada implementação e dimensões do problema. Esse tempo é medido considerando a conclusão do último kernel em todas as GPUs. Dessa forma, mesmo que uma GPU tenha terminado antes, o tempo considerado é da GPU que terminou por último. Ela mostra uma ligeira melhora das estratégias que utilizam um algoritmo de distribuição dos *kernels* entre as máquinas, antes da realização da reordenação dos *kernels*, o que sugere que conseguem fazer o balanceamento da carga de forma mais eficiente.

Balanceamento da Carga

Para avaliar o balanceamento da carga entre as máquinas, foi utilizada a Figura 5.36, que mostra a média do desvio padrão entre as GPUs para cada

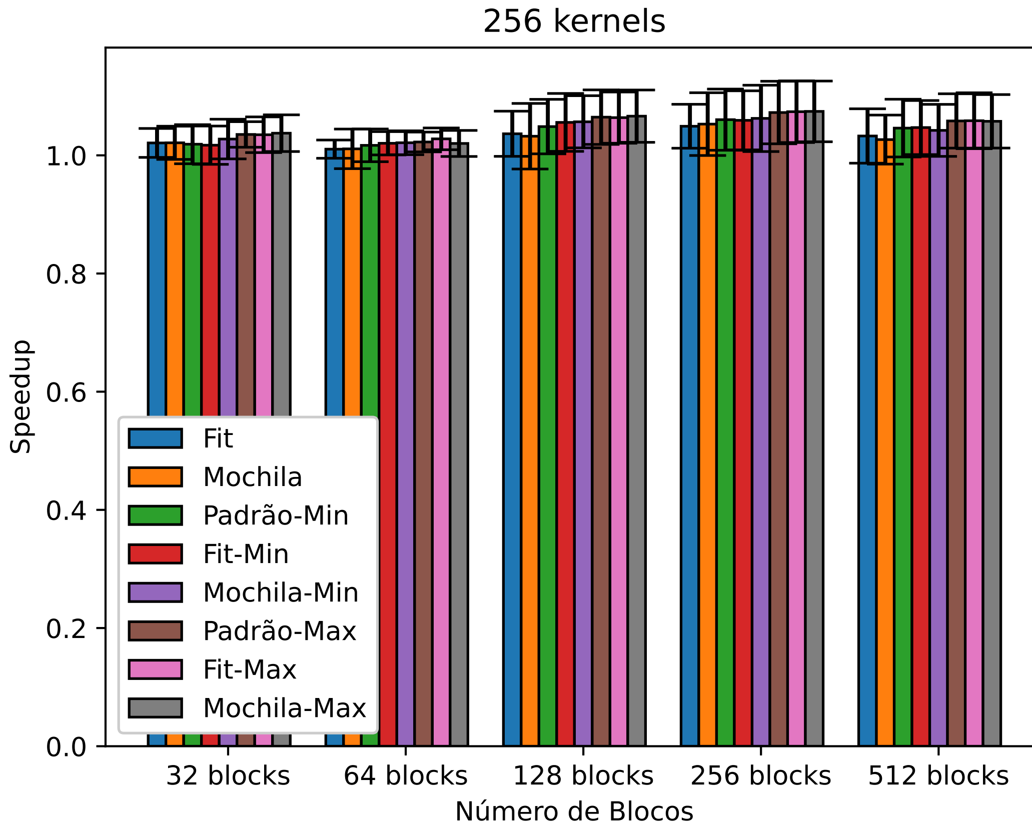


Figura 5.35: *Speedup* das implementações em relação à estratégia padrão, que distribui os *kernels* igualmente entre as GPUs.

uma das estratégias e dimensões do problema. As estratégias que distribuem os *kernels* entre as máquinas sem levar em conta os tempos de execução de cada um são as que apresentam o pior valor para essa métrica. Esse resultado é agravado com o aumento do tamanho dos *kernels*, pois *kernels* maiores costumam demorar mais, já que possuem mais blocos, e uma escolha de escalonamento errada tende a causar maior desbalanceamento.

Quando escalonados *kernels* pequenos as estratégias min-min e max-min apresentaram resultados bastante semelhantes, porém com o aumento do tamanho dos *kernels*, a estratégia max-min se mostrou mais eficaz para escaloná-los. Esse comportamento já é conhecido para essas estratégias, conforme discutido no capítulo 4.

5.10 Conclusões e Trabalhos Futuros

A utilização de mais filas de processamento pode oferecer ganhos de desempenho na execução de múltiplos *kernels* na GPU, porém esse ganho é limitado pela quantidade de recursos dela que cada kernel consome ao executar. Dessa forma, vão existir cenários em que poucas filas de execução serão

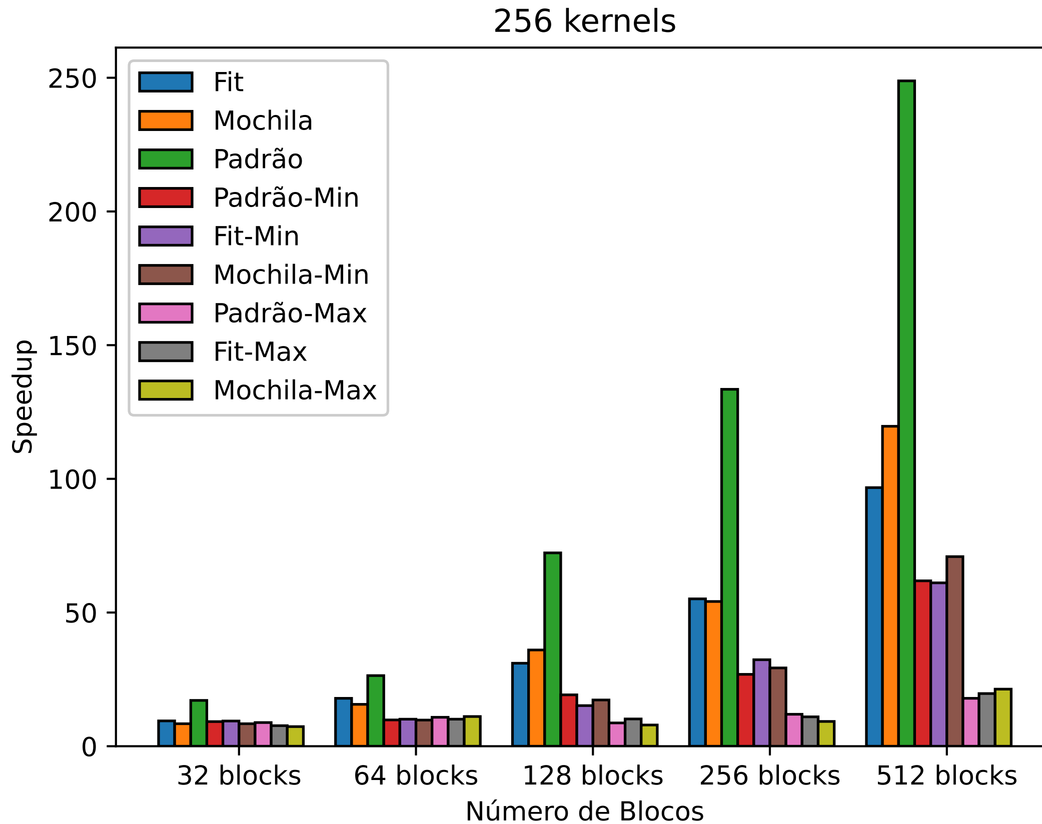


Figura 5.36: *Speedup* das implementações em relação à estratégia padrão, que distribui os *kernels* igualmente entre as GPUs.

suficientes para extrair o máximo do desempenho e vão existir cenários em que serão necessárias todas as 32 filas. Mas mesmo que se tenham *kernels* grandes, que não podem ser executados juntos, a utilização de mais filas pode beneficiar o sistema. Os blocos de um kernel são distribuídos entre os SMs da GPU disponíveis, sendo assim, à medida que um kernel vai chegando ao final de sua execução, os SMs vão sendo liberados pelos blocos que já encerraram. Neste momento, caso haja outra fila de execução da GPU sendo utilizada, um novo kernel pode iniciar nestes SMs liberados.

Quando utilizadas mais de uma fila de execução, a ordem de execução dos *kernels* pode afetar o desempenho individual de cada um deles. Porém, essa ordem não altera de forma significativa o tempo de execução final do sistema. Esse resultado sugere que extrair ganhos de desempenho por meio da reordenação de kernel pode ser muito difícil ou impactar muito pouco o tempo de execução final.

Foram apresentados alguns comportamentos das filas de execução que justificam a dificuldade de se reordenar de forma eficiente, entre eles, a dificuldade de se estimar o tempo de execução de um kernel em um ambiente de filas concorrentes.

Como trabalho futuro, pretende-se adaptar as estratégias para que atualizem o tempo estimado de execução de um kernel de acordo com o número de SMs disponíveis para ele. Com um estimador de tempo mais preciso, espera-se que as reordenações sejam mais precisas.

CAPÍTULO 6

Conclusões e Trabalhos Futuros

As implementações de ordenação segmentada testadas nesse trabalho divergem bastante em seus desempenhos com a mudança nas dimensões do problema, dessa forma, um guia aos programadores com as melhores implementações para as dimensões sendo tratadas pode reduzir o tempo de execução dessa tarefa em mais de 10 vezes. A implementação paralela de ordenação segmentada proposta nesse trabalho, denominada fix sort, mostrou-se eficiente em quase todas as dimensões. Em apenas 6% dos casos o resultado obtido pelo *fix sort* é mais de 50% mais lento que a melhor implementação nesses cenários. Concluímos que o *fix sort* é uma solução bastante consistente para a grande maioria das dimensões avaliadas sem grandes variações de desempenho.

A ordenação segmentada de vetores é parte da solução de vários outros problemas, como é o caso da heurística min-min, utilizada para se obter uma solução rápida e de boa qualidade para o problema de escalonamento de tarefas em ambiente de computação heterogênea. A implementação híbrida (CPU/GPU) desta heurística apresentada neste trabalho obteve *speedups* maiores que 100, quando comparados com implementações sequenciais. Também a comparação com resultados que exploram o paralelismo na implementação do min-min e os algoritmos propostos se mostraram competitivos. Além disso, ela se mostrou uma alternativa ao escalonador nativo da ferramenta Cloudsim, reduzindo o tempo de execução total do sistema em mais de 3 vezes.

O escalonamento de *kernels* no ambiente simulado se mostrou eficiente, principalmente, quando utilizada a heurística min-min para distribuir as tarefas entre as GPUs e, posteriormente, o algoritmo da mochila para reordenar os *kernels*. Enquanto a heurística min-min melhora a distribuição de carga en-

tre as máquinas, a mochila unidimensional otimiza a ordem de execução dos *kernels*. Essas vantagens são maximizadas quando o ambiente é heterogêneo. Como esse ambiente simulado trata-se de uma simplificação do ambiente real, novos testes foram propostos.

Na reordenação de *kernels* no ambiente real, foram mensuradas várias métricas para avaliar o impacto das estratégias nos tempos de execução e no consumo de energia da GPU. A utilização do algoritmo da mochila mostrou ser competitiva para reduzir o tempo de resposta do sistema, porém o problema se mostrou muito mais complexo que no ambiente simulado, devido às variáveis inseridas pelo ambiente real. Como os algoritmos de reordenação necessitam que as estimativas dos tempos de execução dos *kernels* sejam muito precisas, uma estimativa inicial ruim pode impactar negativamente as iterações seguintes dessa estratégia. Cenário que pode ter impactado no desempenho das reordenações.

O estudo de escalonamento de tarefas e reordenação de *kernels* foi ampliado para múltiplas GPUs. Nesses dois casos as heurísticas min-min e max-min se mostraram eficientes na distribuição das tarefas entre as máquinas, tanto no ambiente real como em um ambiente simulado de computação em nuvem.

6.1 Resumo dos Objetivos e Principais Resultados

Vários problemas reais precisam de uma implementação eficiente da ordenação segmentada de vetores. Dessa forma, esse trabalho apresentou a comparação de diversas implementações para esse problema, comparando suas complexidades assintóticas e implementações para entender os pontos fortes e fracos de cada uma. O trabalho mostrou que as implementações em GPU conseguem resolvê-lo de forma eficiente e também apresenta a estratégia *fix sort*, que realiza um pré e um pós-processamento para permitir que algoritmos de segmento único possam ser utilizados nessa tarefa. Em conjunto com o algoritmo de ordenação da biblioteca Thrust, o *fix sort* foi a estratégia que se mostrou mais eficiente na maioria dos cenários do *benchmark* proposto.

Posteriormente, seguindo o algoritmo proposto por Ezzatti et al. (2013), foi apresentada uma implementação linear da heurística min-min utilizando ordenação segmentada de vetores. Testes foram realizados utilizando implementações dessa heurística em CPU, GPU e híbridas (CPU/GPU) e comparados com a implementação tradicional da heurística, bem como com a implementação apresentada por Pedemonte et al. (2016) e mostraram que ambas são competitivas. Os resultados também mostraram que as implementações híbridas, que resolvem a ordenação segmentada em GPU e, posteriormente, o

restante do problema em CPU foram as que apresentaram os melhores resultados.

Com uma implementação eficiente da heurística min-min, testes de escalonamento de tarefas em ambiente de computação heterogênea foram realizados. Para isso utilizou-se o simulador CloudSim para simular um ambiente em nuvem e o escalonador nativo do simulador foi comparado com as heurísticas min-min e max-min. Foram realizados testes em dois cenários diferentes. O primeiro mantinha a relação entre tarefas e máquinas fixa, isto é, quando dobrava o número de tarefas, o número de máquinas também era dobrado. O segundo cenário mantinha o poder computacional do ambiente e o número de tarefas, ou seja, quando o número de máquinas aumentava, o poder computacional de cada uma era diminuído, para manter o mesmo poder computacional do ambiente. Os resultados mostraram que o desvio padrão entre as soluções propostas pelas heurísticas diminui com o aumento da dimensão do problema, mas isso não acontece com o algoritmo padrão do simulador, que mantém um desvio padrão alto em todas as dimensões. E também que os *makespans* das heurísticas são pelo menos 3,6 vezes melhores que o algoritmo padrão do simulador em qualquer dimensão do problema.

Outra abordagem de escalonamento de tarefas consiste da reordenação de *kernels* na GPU. As GPUs atuais permitem a execução concorrente entre *kernels* sempre que exista recursos disponíveis. Dessa forma, foram apresentadas duas estratégias de reordenação de *kernels* em ambiente de computação em nuvem para tentar maximizar a utilização desses recursos. A primeira estratégia consiste em executar os *kernels* na ordem que chegam, enquanto a segunda utiliza o algoritmo da mochila para fazer a seleção dos *kernels* em cada iteração. Os resultados mostraram que a reordenação de *kernels* poderia ser um recurso para otimizar o tempo de execução dos *kernels*. Essas estratégias foram estendidas para um ambiente simulado com múltiplas GPUs, que mostraram que a utilização da heurística min-min em conjunto com a mochila conseguiam reduzir o tempo de conclusão do conjunto de *kernels* sendo executados.

Como o ambiente simulado trata-se de uma simplificação do ambiente real, como, por exemplo, não possui filas de execução nem três dimensões de recursos que precisam ser controlados, testes em um ambiente real também foram realizados. No ambiente com uma única GPU, os resultados mostraram que o algoritmo da mochila pode melhorar o tempo de resposta dos *kernels*, enquanto que no ambiente com múltiplas GPUs, a heurística max-min é a que consegue distribuir melhor a carga de trabalho entre as máquinas. Além disso, foi mostrado também que o ambiente real é muito mais complexo que o ambiente simulado e a importância de um estimador de tempo preciso para

os algoritmos de reordenação.

6.2 Limitações

A execução dos *kernels* em uma GPU real foi realizada usando escalonamento estático, o que pode ser um problema para algumas aplicações reais, principalmente se tratando de um ambiente de computação em nuvem.

As simulações do escalonamento de *kernels* foram realizadas em um ambiente de computação em nuvem que se aproxima do real, mas que contempla somente uma única dimensão de resursos da GPU. Um ambiente real deveria incluir ao menos outras três dimensões: memória compartilhada, número de registradores e número de threads. Porém, os resultados são um indicador de que com um bom algoritmo de escalonamento de kernel, pode-se reduzir o tempo de resposta dos múltiplos *kernels* em execução.

Os algoritmos de reordenação de *kernels* em ambiente real utilizam o tempo de execução sequencial como tempo estimado deles. Conforme apresentado nos resultados, esses tempos podem ser muito impactados pelas execuções concorrentes dos *kernels*, podendo então, ter grande impacto na reordenação dessas estratégias.

6.3 Trabalhos Futuros

Como trabalho futuro, pretende-se implementar um algoritmo de ordenação segmentada de vetores baseado no radix sort, mas que toma como argumento também o vetor de segmentos. Dessa forma, o radix sort poderia levar em conta o índice dos segmentos como parte da ordenação dos elementos do vetor, funcionando de forma semelhante ao *fix sort*, mas sem a etapa de ajuste dos elementos do vetor.

Além disso, pretende-se criar uma nova implementação do *fix sort* que consista em adicionar em cada elemento do vetor o maior elemento multiplicado pelo índice do segmento. Dessa forma, a implementação compreenderia também outros tipos numéricos, como double e float. Outras dimensões para o problema também devem ser testadas, como, por exemplo, vetores com tamanho menores que 32768 e que não sejam múltiplos de 2.

Em relação ao escalonamento de tarefas em um ambiente de computação heterogênea pretende-se utilizar o min-min em conjunto com outras heurísticas, como, por exemplo, um algoritmo genético que busque novas soluções a partir do resultado do min-min. Para comparar as heurísticas serão utilizadas entradas que considerem três propriedades do ambiente: heterogeneidade da máquina, heterogeneidade da tarefa e consistência. Essas novas estratégias

serão testadas também em um ambiente de computação em nuvem.

Sobre a reordenação de *kernels* em uma GPU, pretende-se estudar melhores estimadores de tempo dos *kernels* para serem entregues aos algoritmos já existentes, bem como desenvolver um algoritmo genético que, partindo da mochila unidimensional desenvolvida, busque por melhores soluções de ordenação.

6.4 Publicações Derivadas do Trabalho

O presente trabalho gerou as seguintes publicações:

- **Título**: An Evaluation of Segmented Sorting Strategies on GPUs.
  Autores: Rafael Schmid; Flávia Pisani; Edson Borin; Edson Cáceres.
  Evento: IEEE 18th International Conference on High Performance Computing and Communications (HPCC 2016).
  Ano: 2016.
  Local: Sidney, Austrália.

- **Título**: Implementações Eficientes da Heurística Min-Min para o HCSP em GPU. In: , 2018. Anais do Encontro de Teoria da Computação (ETC).
  Autores: Rafael Schmid; Edson Cáceres.
  Evento: III Encontro de Teoria da Computação (ETC 2018).
  Ano: 2018.
  Local: Natal, Brasil.

- **Título**: Escalonamento de Tarefas usando Implementações Híbridas GPU/CPU das Heurísticas Min-min e Max-min no Cloudsim.
  Autores: Rafael Schmid; Edson Cáceres.
  Evento: II Workshop em Computação Heterogênea (WCH 2018). XIX Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD 2018).
  Ano: 2018.
  Local: São Paulo, Brasil.

- **Título**: Fix Sort: A Good Strategy to Perform Segmented Sorting.
  Autores: Rafael Schmid; Edson Cáceres.
  Evento: 17th International Conference on High Performance Computing & Simulation (HPCS 2019).
  Ano: 2019.
  Local: Dublin, Irlanda.

O trabalho abaixo foi submetido em revista e aguarda avaliação:

- **Título**: An Evaluation of Fast Segmented Sorting Implementations on GPUs.
  Autores: Rafael Schmid; Flávia Pisani; Edson Cáceres; Edson Borin.

Dois trabalhos estão sendo preparados para submissão em eventos:

- Resultados obtidos no Capítulo 4, sobre escalonamento de tarefas em múltiplas máquinas, considerando ambientes homogêneos e heterogêneos.
- Resultados obtidos no Capítulo 5, sobre reordenação de *kernels* em uma única GPU e reordenação de *kernels* em múltiplas GPUs.

Bibliografia

Agarwal, D., Jain, S., et al. (2014). Efficient optimal algorithm of task scheduling in cloud computing environment. *arXiv preprint arXiv:1404.2076*. Citado nas páginas 39 e 57.

Amarís, M., Cordeiro, D., Goldman, A., e d. Camargo, R. Y. (2015). A simple BSP-based model to predict execution time in GPU applications. In *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, páginas 285–294. Citado na página 82.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pagina 483–485, New York, NY, USA. Association for Computing Machinery. Citado na página 82.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., e Stoica, I. (2010). A view of cloud computing. *Communications of the ACM*, 53(4):50–58. Citado na página 55.

Baxter, S. (2013). Segmented Sort and Locality Sort - Modern GPU. Accessed on April 26, 2016. Citado na página 8.

Bergstrom, L. e Reppy, J. (2012). Nested Data-Parallelism on the GPU. *SIGPLAN Not.*, 47(9):247–258. Citado na página 8.

Bhoi, U. e Ramanuj, P. N. (2013). Enhanced max-min task scheduling algorithm in cloud computing. *International Journal of Application or Innovation in Engineering and Management (IJAIEM)*, 2(4):259–264. Citado nas páginas 39 e 62.

Braun, T. D., Siegel, H. J., Beck, N., Bölöni, L. L., Maheswaran, M., Reuther, A. I., Robertson, J. P., Theys, M. D., Yao, B., Hensgen, D., e Freund, R. F. (2001). A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal*

of Parallel and Distributed Computing, 61(6):810 – 837. Citado nas páginas 2, 37, 62, e 63.

Breder, B. (2016). Ordenação da submissão de kernels concorrentes para maximizar a utilização dos recursos da GPU. Master's thesis, Universidade Federal Fluminense. Citado nas páginas xvi, 2, 4, 85, 86, 88, e 90.

Brodtkorb, A. R., Hagen, T. R., e Sætra, M. L. (2013). Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, 73(1):4 – 13. Metaheuristics on GPUs. Citado nas páginas 81, 82, e 84.

Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A. F., e Buyya, R. (2011). Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50. Citado nas páginas 55, 56, 57, e 58.

Calheiros, R. N., Ranjan, R., Rose, C. A. F. D., e Buyya, R. (2009). Cloudsim: A novel framework for modeling and simulation of cloud computing infrastructures and services. Citado nas páginas 56, 57, e 58.

Canabé, M. e Nesmachnow, S. (2012). Parallel implementations of the min-min heterogeneous computing scheduler in GPU. *CLEI Electronic Journal*, 15(3):8–8. Citado na página 39.

Chang, B.-C., Goi, B.-M., Phan, R. C.-W., e Lee, W.-K. (2016). Accelerating multiple precision multiplication in GPU with kepler architecture. In *High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on*, páginas 844–851. IEEE. Citado na página 84.

Clua, E. W. G. e Zamith, M. P. (2015). Programming in CUDA for kepler and maxwell architecture. *Revista de Informática Teórica e Aplicada*, 22(2):233–257. Citado na página 84.

Cruz, R. A., Bentes, C., Breder, B., Vasconcellos, E., Clua, E., de Carvalho, P. M., e Drummond, L. M. (2019). Maximizing the GPU resource usage by reordering concurrent kernels submission. *Concurrency and Computation: Practice and Experience*, 31(18):e4409. Citado nas páginas 81 e 85.

Elzeki, O., Reshad, M., e Elsoud, M. (2012). Improved max-min algorithm in cloud computing. *International Journal of Computer Applications*, 50(12). Citado na página 62.

Etminani, K. e Naghibzadeh, M. (2007). A min-min max-min selective algorihtm for grid task scheduling. In *2007 3rd IEEE/IFIP International Conference in Central Asia on Internet*, páginas 1–7. Citado nas páginas 38 e 62.

Ezzatti, P., Pedemonte, M., e Martín, Á. (2013). An efficient implementation of the min-min heuristic. *Computers & Operations Research*, 40(11):2670–2676. Citado nas páginas 2, 4, 39, 41, 45, 47, e 122.

Ferreira, D. R., Carvalho, P. J., Fernandes, H., e Contributors, J. (2015). Robust regression with CUDA and its application to plasma reflectometry. *Review of Scientific Instruments*, 86(11). Citado na página 1.

Ghorpade, J., Parande, J., Kulkarni, M., e Bawaskar, A. (2012). GpGPU processing in CUDA architecture. *arXiv preprint arXiv:1202.4347*. Citado na página 83.

Gregg, C., Dorn, J., Hazelwood, K., e 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*. Citado na página 84.

Guevara, M., Gregg, C., Hazelwood, K., e Skadron, K. (2009). Enabling task parallelism in the CUDA scheduler. In *Workshop on Programming Models for Emerging Architectures*, volume 9. Citeseer. Citado na página 84.

Hou, K., Liu, W., Wang, H., e Feng, W.-c. (2017). Fast segmented sort on GPUs. In *Proceedings of the International Conference on Supercomputing*, ICS '17, New York, NY, USA. Association for Computing Machinery. Citado nas páginas 7, 9, e 16.

Ibarra, O. H. e Kim, C. E. (1977). Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM (JACM)*, 24(2):280–289. Citado na página 41.

Kokilavani, T., Amalarethinam, D. G., et al. (2011). Load balanced min-min algorithm for static meta-task scheduling in grid computing. *International Journal of Computer Applications*, 20(2):43–49. Citado nas páginas 38, 53, e 62.

Kumar, P. e Verma, A. (2012). Independent task scheduling in cloud computing by improved genetic algorithm. *International Journal of Advanced Research in Computer Science and Software Engineering*, 2(5). Citado nas páginas 38, 62, e 63.

Leung, J. Y. (2004). *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press. Citado na página 35.

Li, K., Xu, G., Zhao, G., Dong, Y., e Wang, D. (2011). Cloud task scheduling based on load balancing ant colony optimization. In *Chinagrid Conference (ChinaGrid), 2011 Sixth Annual*, páginas 3–9. IEEE. Citado nas páginas xvi, 38, e 59.

Maheswaran, M., Ali, S., Siegal, H. J., Hensgen, D., e Freund, R. F. (1999). Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings. Eighth*, páginas 30–44. Citado na página 37.

Massobrio, R., Dorronsoro, B., e Nesmachnow, S. (2018). Virtual savant for the heterogeneous computing scheduling problem. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*, páginas 821–827. IEEE. Citado nas páginas 40 e 53.

Min-You Wu, Wei Shu, e Zhang, H. (2000). Segmented min-min: a static mapping algorithm for meta-tasks on heterogeneous computing systems. In *Proceedings 9th Heterogeneous Computing Workshop (HCW 2000) (Cat. No.PR00556)*, páginas 375–385. Citado nas páginas 2 e 37.

Nesmachnow, S. e Canabé, M. (2011). GPU implementations of scheduling heuristics for heterogeneous computing environments. In *XVII Congreso Argentino de Ciencias de la Computación*. Citado nas páginas xv, 35, 38, 39, e 44.

Nesmachnow, S., Cancela, H., e Alba, E. (2010). Heterogeneous computing scheduling with evolutionary algorithms. *Soft Computing*, 15(4):685–701. Citado na página 38.

NVIDIA (2016). CUB: Main Page. Accessed on April 28, 2016. Citado na página 8.

NVIDIA (2019). *Data Center GPU Manager User Guide*. NVIDIA. Citado na página 92.

NVIDIA, C. (2012). Nvidia's next generation CUDA compute architecture: Kepler gk110. *Whitepaper (2012)*. Citado na página 85.

Pai, S., Thazhuthaveetil, M. J., e Govindarajan, R. (2013). Improving gpGPU concurrency with elastic kernels. In *ACM SIGPLAN Notices*, volume 48, páginas 407–418. ACM. Citado na página 85.

Patel, G., Mehta, R., e Bhoi, U. (2015). Enhanced load balanced min-min algorithm for static meta task scheduling in cloud computing. *Procedia Computer Science*, 57(Supplement C):545 – 553. 3rd International Conference

on Recent Trends in Computing 2015 (ICRTC-2015). Citado nas páginas 40 e 53.

Pedemonte, M., Ezzatti, P., e Martín, Á. (2016). Accelerating the min-min heuristic. In *Parallel Processing and Applied Mathematics*, páginas 101–110. Springer. Citado nas páginas 40, 41, 48, 51, e 122.

Peters, H., Köper, M., e Luttenberger, N. (2010). Efficiently using a CUDA-enabled GPU as shared resource. In *2010 10th IEEE International Conference on Computer and Information Technology*, páginas 1122–1127. IEEE. Citado na página 84.

Pisani, F., Pedronette, D. C. G., da S. Torres, R., e Borin, E. (2015). Improving the Performance of the Contextual Spaces Re-Ranking Algorithm on Heterogeneous Systems. In *Proc. XVI Simpósio de Sistemas Computacionais de Alto Desempenho (WSCAD'2015)*, páginas 132–143. Citado na página 1.

Rajput, S. S. e Kushwah, V. S. (2016). A genetic based improved load balanced min-min task scheduling algorithm for load balancing in cloud computing. In *2016 8th international conference on Computational Intelligence and Communication Networks (CICN)*, páginas 677–681. IEEE. Citado nas páginas 40 e 53.

Ravi, V. T., Becchi, M., Agrawal, G., e 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*, páginas 217–228. Citado na página 85.

Satish, N., Kim, C., Chhugani, J., Nguyen, A. D., Lee, V. W., Kim, D., e Dubey, P. (2010). Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort. In *Proc. 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'201)*, SIGMOD '10, páginas 351–362, New York, NY, USA. ACM. Citado na página 15.

Schmid, R. e Cáceres, E. (2019). Fix sort: A good strategy to perform segmented sorting. In *2019 IEEE 17th International Conference on High Performance Computing Simulation (HPCS)*. IEEE. Citado nas páginas 3, 7, 8, 17, e 34.

Schmid, R., Pisani, F., Borin, E., e Cáceres, E. (2016). An evaluation of segmented sorting strategies on GPUs. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, páginas 1123–1130. Citado nas páginas 8 e 31.

Tanasic, I., Gelado, I., Cabezas, J., Ramirez, A., Navarro, N., e Valero, M. (2014). Enabling preemptive multiprogramming on GPUs. *ACM SIGARCH Computer Architecture News*, 42(3):193–204. Citado na página 85.

Tsai, C.-W., Huang, W.-C., Chiang, M.-H., Chiang, M.-C., e Yang, C.-S. (2014). A hyper-heuristic scheduling algorithm for cloud. *IEEE Transactions on Cloud Computing*, 2(2):236–250. Citado nas páginas 40 e 63.

Ullman, J. D. (1975). Np-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393. Citado nas páginas 2 e 37.

Wang, G., Lin, Y., e Yi, W. (2010). Kernel fusion: An effective method for better power efficiency on multithreaded GPU. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, páginas 344–350. IEEE. Citado na página 84.

Wang, L., Baxter, S., e Owens, J. D. (2015). Fast Parallel Suffix Array on the GPU. In Träff, L. J., Hunold, S., e Versaci, F., editors, *Euro-Par 2015: Parallel Processing*, páginas 573–587. Springer Berlin Heidelberg, Berlin, Heidelberg. Citado na página 1.

Wu, B., Zhao, Z., Zhang, E. Z., Jiang, Y., e Shen, X. (2013). Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, páginas 57–68, New York, NY, USA. ACM. Citado nas páginas 82 e 83.

Wu, M.-Y. e Shu, W. (2001). A high-performance mapping algorithm for heterogeneous computing systems. In *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, páginas 6 pp.–. Citado na página 37.

Zhong, J. e He, B. (2013). Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1522–1532. Citado na página 85.

APÊNDICE A

Apêndice

A Tabela A.1 apresenta o tempo de execução das estratégias do Capítulo 5 para cada dimensão do problema.

Tabela A.1: Tempo de execução das estratégias para cada dimensão do problema.

Kernels	Blocos	Coeficiente	Implementação	Média	Desv.Padr.
32	32	0.8	Sequencial	173.99	31.66
32	32	0.8	Padrão32	74.75	21.76
32	32	0.8	Fit32	74.02	22.97
32	32	0.8	Mochila32	74.15	21.74
32	32	0.2	Sequencial	245.49	35.50
32	32	0.2	Padrão32	165.27	25.56
32	32	0.2	Fit32	163.58	25.37
32	32	0.2	Mochila32	164.88	25.31
32	32	1.0	Sequencial	235.38	44.77
32	32	1.0	Padrão32	148.14	35.87
32	32	1.0	Fit32	146.98	36.15
32	32	1.0	Mochila32	148.69	35.93
32	64	0.8	Sequencial	218.83	21.71
32	64	0.8	Padrão32	140.19	25.75
32	64	0.8	Fit32	140.15	25.37
32	64	0.8	Mochila32	140.35	23.57
32	64	0.2	Sequencial	432.25	52.71
32	64	0.2	Padrão32	349.17	52.01

Tabela A.1: Tempo de execução das estratégias para cada dimensão do problema.

Kernels	Blocos	Coeficiente	Implementação	Média	Desv.Padr.
32	64	0.2	Fit32	350.18	52.14
32	64	0.2	Mochila32	351.35	53.68
32	64	1.0	Sequencial	358.36	66.01
32	64	1.0	Padrão32	279.10	53.26
32	64	1.0	Fit32	279.33	53.60
32	64	1.0	Mochila32	279.71	53.92
32	128	0.8	Sequencial	335.08	29.43
32	128	0.8	Padrão32	258.53	25.43
32	128	0.8	Fit32	257.97	27.27
32	128	0.8	Mochila32	258.30	26.51
32	128	0.2	Sequencial	843.55	43.55
32	128	0.2	Padrão32	753.43	42.48
32	128	0.2	Fit32	754.39	43.27
32	128	0.2	Mochila32	754.23	43.38
32	128	1.0	Sequencial	562.70	71.52
32	128	1.0	Padrão32	483.32	67.37
32	128	1.0	Fit32	481.65	67.39
32	128	1.0	Mochila32	482.29	68.53
32	256	0.8	Sequencial	594.28	64.95
32	256	0.8	Padrão32	515.58	66.41
32	256	0.8	Fit32	515.20	65.17
32	256	0.8	Mochila32	515.89	64.75
32	256	0.2	Sequencial	1519.32	206.41
32	256	0.2	Padrão32	1423.87	192.90
32	256	0.2	Fit32	1423.63	193.58
32	256	0.2	Mochila32	1422.74	189.54
32	256	1.0	Sequencial	1141.17	130.48
32	256	1.0	Padrão32	1054.70	128.89
32	256	1.0	Fit32	1053.53	129.79
32	256	1.0	Mochila32	1053.91	128.99
32	512	0.8	Sequencial	1285.35	148.47
32	512	0.8	Padrão32	1208.33	139.54
32	512	0.8	Fit32	1208.14	139.62
32	512	0.8	Mochila32	1208.27	139.22
32	512	0.2	Sequencial	3204.63	258.82
32	512	0.2	Padrão32	3108.23	253.79

Tabela A.1: Tempo de execução das estratégias para cada dimensão do problema.

Kernels	Blocos	Coeficiente	Implementação	Média	Desv.Padr.
32	512	0.2	Fit32	3107.23	253.71
32	512	0.2	Mochila32	3107.47	253.15
32	512	1.0	Sequencial	2096.04	147.12
32	512	1.0	Padrão32	2006.87	142.90
32	512	1.0	Fit32	2006.57	143.22
32	512	1.0	Mochila32	2008.64	141.74
64	32	0.8	Sequencial	345.59	58.39
64	32	0.8	Padrão32	141.61	37.46
64	32	0.8	Fit32	140.27	38.69
64	32	0.8	Mochila32	140.59	38.51
64	32	0.2	Sequencial	533.74	46.28
64	32	0.2	Padrão32	367.09	44.04
64	32	0.2	Fit32	365.25	42.87
64	32	0.2	Mochila32	364.87	42.48
64	32	1.0	Sequencial	417.67	43.00
64	32	1.0	Padrão32	246.37	26.04
64	32	1.0	Fit32	243.36	25.76
64	32	1.0	Mochila32	245.02	25.75
64	64	0.8	Sequencial	405.82	27.00
64	64	0.8	Padrão32	240.74	22.55
64	64	0.8	Fit32	238.75	21.53
64	64	0.8	Mochila32	238.93	22.08
64	64	0.2	Sequencial	854.37	97.43
64	64	0.2	Padrão32	687.43	68.90
64	64	0.2	Fit32	686.54	69.00
64	64	0.2	Mochila32	687.74	68.70
64	64	1.0	Sequencial	670.88	73.05
64	64	1.0	Padrão32	516.40	81.94
64	64	1.0	Fit32	515.38	80.53
64	64	1.0	Mochila32	515.49	80.82
64	128	0.8	Sequencial	668.74	54.66
64	128	0.8	Padrão32	511.82	46.96
64	128	0.8	Fit32	510.52	45.72
64	128	0.8	Mochila32	510.47	46.31
64	128	0.2	Sequencial	1662.24	106.92
64	128	0.2	Padrão32	1477.68	98.90

Tabela A.1: Tempo de execução das estratégias para cada dimensão do problema.

Kernels	Blocos	Coeficiente	Implementação	Média	Desv.Padr.
64	128	0.2	Fit32	1475.62	96.44
64	128	0.2	Mochila32	1476.37	100.45
64	128	1.0	Sequencial	1163.44	185.14
64	128	1.0	Padrão32	1007.55	167.89
64	128	1.0	Fit32	1007.36	166.95
64	128	1.0	Mochila32	1007.42	168.12
64	256	0.8	Sequencial	1157.68	166.87
64	256	0.8	Padrão32	996.64	159.08
64	256	0.8	Fit32	995.73	156.57
64	256	0.8	Mochila32	995.32	156.72
64	256	0.2	Sequencial	2850.49	295.69
64	256	0.2	Padrão32	2680.84	283.65
64	256	0.2	Fit32	2680.66	282.47
64	256	0.2	Mochila32	2680.16	284.16
64	256	1.0	Sequencial	2057.84	225.87
64	256	1.0	Padrão32	1903.35	216.69
64	256	1.0	Fit32	1903.91	215.98
64	256	1.0	Mochila32	1904.87	217.40
64	512	0.8	Sequencial	2116.57	64.97
64	512	0.8	Padrão32	1960.36	74.12
64	512	0.8	Fit32	1960.37	76.46
64	512	0.8	Mochila32	1960.03	72.96
64	512	0.2	Sequencial	5753.80	579.95
64	512	0.2	Padrão32	5568.57	573.37
64	512	0.2	Fit32	5568.44	574.98
64	512	0.2	Mochila32	5569.42	576.22
64	512	1.0	Sequencial	4338.28	629.89
64	512	1.0	Padrão32	4159.43	612.21
64	512	1.0	Fit32	4160.24	610.41
64	512	1.0	Mochila32	4162.12	613.68
128	32	0.8	Sequencial	685.04	69.25
128	32	0.8	Padrão32	281.96	46.43
128	32	0.8	Fit32	280.56	48.62
128	32	0.8	Mochila32	281.13	47.21
128	32	0.2	Sequencial	1049.27	57.80
128	32	0.2	Padrão32	714.72	47.04

Tabela A.1: Tempo de execução das estratégias para cada dimensão do problema.

Kernels	Blocos	Coeficiente	Implementação	Média	Desv.Padr.
128	32	0.2	Fit32	712.02	46.08
128	32	0.2	Mochila32	711.71	46.80
128	32	1.0	Sequencial	847.83	57.34
128	32	1.0	Padrão32	493.26	59.32
128	32	1.0	Fit32	490.85	60.28
128	32	1.0	Mochila32	491.10	59.77
128	64	0.8	Sequencial	885.77	41.95
128	64	0.8	Padrão32	536.09	27.38
128	64	0.8	Fit32	535.65	27.03
128	64	0.8	Mochila32	534.24	26.70
128	64	0.2	Sequencial	1763.38	182.88
128	64	0.2	Padrão32	1432.66	178.01
128	64	0.2	Fit32	1430.99	177.68
128	64	0.2	Mochila32	1432.26	178.56
128	64	1.0	Sequencial	1269.43	145.35
128	64	1.0	Padrão32	955.25	118.44
128	64	1.0	Fit32	954.64	118.23
128	64	1.0	Mochila32	953.75	118.78
128	128	0.8	Sequencial	1441.08	73.01
128	128	0.8	Padrão32	1116.53	83.35
128	128	0.8	Fit32	1113.80	82.42
128	128	0.8	Mochila32	1113.77	83.75
128	128	0.2	Sequencial	3116.40	245.81
128	128	0.2	Padrão32	2783.24	239.32
128	128	0.2	Fit32	2777.45	241.32
128	128	0.2	Mochila32	2777.94	241.22
128	128	1.0	Sequencial	2296.75	134.51
128	128	1.0	Padrão32	1960.32	147.56
128	128	1.0	Fit32	1960.58	146.10
128	128	1.0	Mochila32	1961.16	146.99
128	256	0.8	Sequencial	2551.37	224.90
128	256	0.8	Padrão32	2221.84	213.25
128	256	0.8	Fit32	2221.30	213.37
128	256	0.8	Mochila32	2221.60	214.39
128	256	0.2	Sequencial	6003.48	149.21
128	256	0.2	Padrão32	5650.35	136.60

Tabela A.1: Tempo de execução das estratégias para cada dimensão do problema.

Kernels	Blocos	Coeficiente	Implementação	Média	Desv.Padr.
128	256	0.2	Fit32	5650.51	137.96
128	256	0.2	Mochila32	5651.27	136.69
128	256	1.0	Sequencial	4330.98	395.32
128	256	1.0	Padrão32	3976.36	379.92
128	256	1.0	Fit32	3976.49	381.77
128	256	1.0	Mochila32	3974.97	379.61
128	512	0.8	Sequencial	4513.03	505.49
128	512	0.8	Padrão32	4175.54	480.30
128	512	0.8	Fit32	4176.28	481.07
128	512	0.8	Mochila32	4175.01	478.65
128	512	0.2	Sequencial	11097.83	511.91
128	512	0.2	Padrão32	10744.41	514.99
128	512	0.2	Fit32	10746.61	516.89
128	512	0.2	Mochila32	10745.49	512.49
128	512	1.0	Sequencial	8170.39	822.29
128	512	1.0	Padrão32	7822.92	826.38
128	512	1.0	Fit32	7821.28	825.61
128	512	1.0	Mochila32	7819.25	824.04
256	32	0.8	Sequencial	1374.56	27.09
256	32	0.8	Padrão32	539.07	29.45
256	32	0.8	Fit32	537.72	29.76
256	32	0.8	Mochila32	535.83	30.38
256	32	0.2	Sequencial	2157.15	161.43
256	32	0.2	Padrão32	1471.09	125.06
256	32	0.2	Fit32	1466.65	123.93
256	32	0.2	Mochila32	1469.22	124.63
256	32	1.0	Sequencial	1692.19	101.53
256	32	1.0	Padrão32	1021.39	82.52
256	32	1.0	Fit32	1018.03	81.86
256	32	1.0	Mochila32	1018.54	80.44
256	64	0.8	Sequencial	1783.99	105.30
256	64	0.8	Padrão32	1058.48	79.50
256	64	0.8	Fit32	1056.74	78.46
256	64	0.8	Mochila32	1057.12	79.19
256	64	0.2	Sequencial	3479.92	168.28
256	64	0.2	Padrão32	2833.41	147.39

Tabela A.1: Tempo de execução das estratégias para cada dimensão do problema.

Kernels	Blocos	Coeficiente	Implementação	Média	Desv.Padr.
256	64	0.2	Fit32	2830.71	146.68
256	64	0.2	Mochila32	2832.16	144.88
256	64	1.0	Sequencial	2688.99	159.26
256	64	1.0	Padrão32	2021.81	126.41
256	64	1.0	Fit32	2019.29	125.41
256	64	1.0	Mochila32	2020.30	125.05
256	128	0.8	Sequencial	2818.04	121.96
256	128	0.8	Padrão32	2159.99	107.24
256	128	0.8	Fit32	2156.13	105.12
256	128	0.8	Mochila32	2157.49	105.06
256	128	0.2	Sequencial	6097.77	365.32
256	128	0.2	Padrão32	5442.41	327.98
256	128	0.2	Fit32	5439.01	326.16
256	128	0.2	Mochila32	5439.24	326.67
256	128	1.0	Sequencial	4585.40	234.43
256	128	1.0	Padrão32	3937.95	219.26
256	128	1.0	Fit32	3934.28	218.66
256	128	1.0	Mochila32	3933.98	217.86
256	256	0.8	Sequencial	4854.85	212.26
256	256	0.8	Padrão32	4201.67	217.13
256	256	0.8	Fit32	4201.47	219.81
256	256	0.8	Mochila32	4201.12	218.17
256	256	0.2	Sequencial	12045.56	248.63
256	256	0.2	Padrão32	11326.81	235.23
256	256	0.2	Fit32	11326.72	231.46
256	256	0.2	Mochila32	11326.76	229.51
256	256	1.0	Sequencial	8133.07	661.87
256	256	1.0	Padrão32	7455.40	631.56
256	256	1.0	Fit32	7452.22	631.39
256	256	1.0	Mochila32	7447.98	628.21
256	512	0.8	Sequencial	9432.68	600.22
256	512	0.8	Padrão32	8755.75	594.81
256	512	0.8	Fit32	8758.25	599.71
256	512	0.8	Mochila32	8751.81	594.10
256	512	0.2	Sequencial	23003.99	1676.40
256	512	0.2	Padrão32	22293.25	1634.99

Tabela A.1: Tempo de execução das estratégias para cada dimensão do problema.

Kernels	Blocos	Coeficiente	Implementação	Média	Desv.Padr.
256	512	0.2	Fit32	22287.56	1640.93
256	512	0.2	Mochila32	22285.65	1639.55
256	512	1.0	Sequencial	15591.84	1047.23
256	512	1.0	Padrão32	14908.19	1039.91
256	512	1.0	Fit32	14911.25	1038.43
256	512	1.0	Mochila32	14908.39	1041.85