PROCESSAMENTO PARALELO EM CUDA APLICADO AO MODELO DE GERAÇÃO DE CENÁRIOS SINTÉTICOS DE VAZÕES E ENERGIAS - GEVAZP

André Emanoel Rabello Quadros

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia Elétrica.

Orientadora: Carmen Lucia Tancredo Borges

Rio de Janeiro

Março de 2016

PROCESSAMENTO PARALELO EM CUDA APLICADO AO MODELO DE GERAÇÃO DE CENÁRIOS SINTÉTICOS DE VAZÕES E ENERGIAS - GEVAZP

André Emanoel Rabello Quadros

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Examinada por:

_____

Prof[a]. Carmen Lucia Tancredo Borges, D.Sc.

_____

Prof. Antônio Carlos Siqueira de Lima, D.Sc.

_____

Dra. Maria Elvira Piñeiro Maceira, D.Sc.

_____

Prof. Sérgio Barbosa Villas-Boas, Ph.D.

RIO DE JANEIRO, RJ – BRASIL

MARÇO DE 2016

Quadros, André Emanoel Rabello

Processamento Paralelo em CUDA Aplicada ao Modelo de Geração de Cenários Sintéticos de Vazões e Energias – GEVAZP / André Emanoel Rabello Quadros. – Rio de Janeiro: UFRJ/COPPE, 2016.

XII, 101 p.: il.; 29,7 cm.

Orientador(a): Carmen Lucia Tancredo Borges

Dissertação (mestrado) – UFRJ/ COPPE/ Programa de Engenharia Elétrica, 2016.

Referências Bibliográficas: p. 96-99.

1. Programação Paralela. 2. GPU. 3. CUDA. 4. Planejamento da Operação Energética. 5. Geração de cenários sintéticos. I. Borges, Carmen Lucia Tancredo. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

Aos amores da minha vida: minha filha Isabela e minha esposa Andréa.

Às minhas duas mães, Dona Sônia e vovó Nira.

AGRADECIMENTOS

Primeiramente agradeço à UFRJ pela oportunidade de estudar em uma das melhores instituições de ensino e pesquisa do nosso país.

À minha orientadora Carmen, pelos conhecimentos compartilhados, pela paciência, incentivo e por ter me acompanhado ao longo deste trabalho.

Agradeço ao CEPEL e em especial à Maria Elvira pelo incentivo e oportunidade de realizar esta pesquisa.

Aos professores Sérgio Barbosa Villas-Boas e Antônio Carlos Siqueira de Lima por aceitarem o convite de participar da banca.

Agradeço ao Roberto Pinto pelos ensinamentos em computação paralela, em mpi, e pelas horas que esteve ao meu lado. Sou muito grato por sua dedicação.

Aos amigos Felipe Machado e Fábio Lares pela generosidade em compartilhar seus conhecimentos e por sempre me incentivarem nas horas mais difíceis.

Aos amigos André Diniz, Débora e Ricardo Dutra pelos conhecimentos transmitidos e pela ajuda essencial nas revisões.

Quero agradecer a toda minha família pelo amor e carinho que recebo de todos. Em especial, gostaria de agradecer à minha mãe e minha avó Nira, que são os alicerces da minha vida e aos meus irmãos Angélica e Yuri por tudo que representam em minha vida.

Um agradecimento especial à minha filha Isabela que sempre me motivou a seguir em frente com seu amor e sua alegria contagiante, e aquele sorriso que tranquiliza. Você é o meu orgulho filha.

Quero agradecer muitíssimo a minha esposa Andréa, pelo amor, carinho, paciência e pela compreensão que teve durante essa jornada. Continuaremos de mãos dadas. Obrigado.

A todos os amigos do CEPEL, em especial Valk, Ana Carolina, César, Ricardo, Amauri, Flávia, Bruno, Tatiana, Cristiane, Léo e Pamella pela amizade sincera e pelas boas risadas. E a todos que participaram dessa conquista.

E por fim agradeço a Deus por tudo que me deu em minha vida.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

PROCESSAMENTO PARALELO EM CUDA APLICADO AO MODELO DE GERAÇÃO DE CENÁRIOS SINTÉTICOS DE VAZÕES E ENERGIAS - GEVAZP

André Emanoel Rabello Quadros

Março/2016

Orientador: Carmen Lucia Tancredo Borges

Programa: Engenharia Elétrica

No problema de planejamento da operação hidrotérmica, a função do custo futuro da operação é calculada considerando a incerteza das afluências futuras. A geração de cenários sintéticos de afluência é utilizada por modelos de otimização do planejamento de médio e curto prazo, que percorrem uma árvore de afluências definindo a alocação ótima do uso da água dos reservatórios e da geração térmica, ao longo do período de estudo. Este trabalho apresenta uma metodologia de programação paralela em GPU, utilizando a plataforma de programação CUDA, aplicada ao modelo de geração de cenários sintéticos de vazões e energias – GEVAZP, que é utilizado oficialmente no sistema brasileiro. A proposta atuou no processo de agrupamento dos cenários. Na solução em CUDA, as distâncias entre as vazões e os centróides dos grupos são calculadas em paralelo na GPU, e seu resultado utilizado no algoritmo de agrupamento, na CPU. Explorando a independência entre os cenários pertencentes a um mesmo período, foi proposta também uma solução em MPI, aplicando técnicas de computação paralela em *cluster* de computadores, dividindo-se a tarefas da geração de cenários entre os processadores do *cluster*. As propostas em MPI e CUDA foram integradas em uma solução híbrida, se beneficiando das duas abordagens, aumentando o ganho de desempenho final. Os resultados obtidos em casos reais mostram que a solução paralela obteve ganhos de desempenho expressivos, apresentando redução do tempo computacional de até 42 vezes em relação à versão serial.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

PARALLEL PROCESSING IN CUDA APPLIED TO SYNTHETIC STREAMFLOW AND ENERGIES SCENARIO MODEL - GEVAZP

André Emanoel Rabello Quadros
March/2016

Advisor: Carmen Lucia Tancredo Borges

Department: Electrical Engineering

In the hydrothermal operation planning problem, the value of the cost-to-go function of the system operation is calculated considering the uncertainty on future inflows to reservoirs. The generation of synthetic streamflow scenarios is used for optimization models for the mid and short-term planning, which traverse a streamflow tree defining the optimal allocation of the use of water reservoir and thermal generation along the study horizon. This work presents a parallel programming methodology in GPU, using CUDA parallel computing platform, applied to the synthetic streamflow and energies scenario model – GEVAZP, that is officially used in the Brazilian system. The proposed approach was applied on the scenario clustering process. On the CUDA solution, the distances between the streamflow and the centroids of groups are calculated in parallel on the GPU, and such results are used in the clustering algorithm, on the CPU. Based on the independence of the scenarios in the same time period, a solution in MPI has also been proposed, using parallel computing techniques in a computer cluster, by dividing the tasks of generation scenarios to the cluster processors. The proposals in MPI and CUDA have been integrated into a hybrid solution, taking advantage of both approaches by increasing the gain of the final performance. The results of real cases show that the parallel solution yielded significant performance gains, reducing the computational time up to 42 times compared to the serial version.

SUMÁRIO

1 INTRODUÇÃO 1

1.1 Considerações iniciais 1

1.2 Organização do trabalho 3

2 ARQUITETURA DAS GPUS NVIDIA E O MODELO DE PROGRAMAÇÃO CUDA 5

2.1 Computação Paralela 5

2.2 Arquitetura das GPUs NVIDIA 10

2.2.1 Graphics Processing Unit - GPU 11

2.2.2 Stream Multiprocessor e paradigma SIMT 14

2.2.3 Warps 15

2.2.4 Categorias das GPUs NVIDIA 16

2.3 Modelo de Programação CUDA 20

2.3.1 Kernels 20

2.3.2 Hierarquia de *Threads* 23

2.3.3 Hierarquia de acesso à memória 26

2.3.4 Modelo de escalabilidade da GPU 28

2.3.5 Nível de ocupação do *Stream Multiprocessor* 29

3 PARALELIZAÇÃO DO MODELO DE GERAÇÃO DE CENÁRIOS SINTÉTICOS - GEVAZP 30

3.1 Planejamento da Operação de Sistemas Hidrotérmicos 30

3.2 Otimização em Sistemas Hidrotérmicos 33

3.3 Cadeia de Modelos do Planejamento da Operação do Sistema Brasileiro 37

3.4 Geração de séries sintéticas para os modelos NEWAVE, SUISHI e DECOMP 40

3.5 Modelo Autorregressivo Periódico 41

3.6 Agrupamento dos cenários hidrológicos 42

3.7 Estratégias de Paralelização do modelo GEVAZP 45

3.7.1 Estudo da metodologia do problema 45

3.7.2 Estudo do Algoritmo de Agregação 46

3.7.3 Modificações no Algoritmo de Hartigan & Wong 47

3.7.4 Testes de validação do algoritmo de agrupamento modificado 48

3.8 Particionamento do problema, granularidade e modelo de paralelização para cada parte do problema 51

3.8.1 Estratégia 1 – Utilização da biblioteca MPI 52

3.8.2 Estratégia 2 – Utilização da plataforma CUDA 53

4 GEVAZP PARALELO EM CUDA 56

4.1 Implementação da solução paralela em CUDA 57
4.2 Gevazp_gm – Utilização de Memória Global 60
4.2.1 Definição de blocos e threads 61
4.2.2 Otimizações do código fonte em CUDA 63
4.3 Gevazp_sh – Utilização de Memória Compartilhada 65
4.3.1 Definição de blocos e *threads* 69
4.4 Gevazp_cublas – Utilização da biblioteca CUBLAS 73
4.5 Implementação da solução paralela utilizando MPI 75
4.6 Implementação da solução paralela híbrida utilizando MPI e CUDA 76

5 EXPERIMENTOS E RESULTADOS 79

5.1 Experimentos da etapa de inicialização 80
5.2 Resultados da rotina OPTRA 86
5.3 Resultados da agregação de cenários e overall speedup 88
5.4 Resultados para um caso com 6000 aberturas 89
5.5 Resultados para um caso com 6000 aberturas com a solução híbrida MPI e CUDA 90

6 CONCLUSÃO E TRABALHOS FUTUROS 93

6.1 Trabalhos futuros 94

REFERÊNCIAS BIBLIOGRÁFICAS 96

ÍNDICE DE FIGURAS

Figura 2-1: Arquitetura de computadores de Von Newman [8] 5
Figura 2-2: Arquiteturas de memória compartilhada, (à esq.) e distribuída (à dir.) [8] 6
Figura 2-3: GPU tem milhares de cores [10] 10
Figura 2-4: Como a GPU acelera as aplicações [11] 11
Figura 2-5: Operações de ponto flutuante por segundo em CPU e GPU (GFLOP/s) [5] 12
Figura 2-6: Largura de banda de acesso à memória para CPU e GPU [5] 12
Figura 2-7: Comparação entre as arquiteturas da CPU e GPU [5] 13
Figura 2-8: Detalhes dos CUDA-cores, do Stream Multiprocessor e da GPU [5] 14
Figura 2-9: Cada SMx Kepler possui 4 *warp-scheduler* e 8 *instruction dispatch unit* [12] 15
Figura 2-2-10: SMx - 192 CUDA-cores de precisão simples e 64 de precisão dupla [12] 18
Figura 2-11: Paralelismo dinâmico [12] 19
Figura 2-12: Hyper-Q permite que até 32 *streams* de diferentes processadores [12] 19
Figura 2-13: Definição e execução de um kernel 21
Figura 2-14: Fluxo de execução de um kernel 22
Figura 2-15: Organização das *threads* em Grids e Blocos 24
Figura 2-16: Mapeamento das threads para os processadores CUDA [15] 24
Figura 2-17: *Kernel* que soma dois vetores na GPU 26
Figura 2-18: Hierarquia de Memória do modelo de programação CUDA [5] 27
Figura 2-19: Acesso à Memória - Hardware da GPU [15] 27
Figura 2-20: Execução de 8 blocos em GPUs com 2 SMs e 4 SMs [5] 28
Figura 3-1. Esquema de decisões operativas em sistemas hidrotérmicos [21] 33
Figura 3-2. Funções de Custo Presente e Custo Futuro 35
Figura 3-3: Uso ótimo da água 36
Figura 3-4: Horizontes do planejamento da operação 37
Figura 3-5: Cadeia de modelos utilizados no planejamento da expansão 39
Figura 3-6: Geração de Séries - (a) em Paralelo, (b) em Árvore 40
Figura 3-7: Afluências independentes no tempo 45
Figura 3-8: Comparação entre as médias das vazões da versão original e da modificada 49
Figura 3-9: Comparação entre os desvios padrão das vazões versão original e da modificada 49
Figura 3-10. Paralelização em MPI 53
Figura 3-11: Paralelização em CUDA 54
Figura 4-1: Parte do código fonte da função cuda_mahala() 58
Figura 4-2: Passos para cálculo das distâncias de Mahalanobis 60

Figura 4-3: Distribuição das *threads* utilizando-se memória global _____ 61
Figura 4-4: Definição do grid e blocos. Blocos de 1 x 1024 *threads*. *Grid* de 1 x 1 blocos _ 62
Figura 4-5: Estrutura do cálculo das distâncias utilizando-se memória global _____ 63
Figura 4-6: Código fonte em CUDA do kernel distanciaMahalanobis() _____ 64
Figura 4-7: Distribuição das *threads* utilizando-se memória compartilhada _____ 67
Figura 4-8: *Parallel Reduction* em memória compartilhada [38] _____ 68
Figura 4-9: Alocação de memória das estruturas da versão Gevazp_sm _____ 69
Figura 4-10: Definição de *grid* e blocos. Blocos de 64 x 16 *threads*. Grid de 1 x 64 blocos 70
Figura 4-11: Algoritmo do cálculo das distâncias utilizando-se memória compartilhada ___ 72
Figura 4-12: Alocação de memória e utilização da biblioteca CUBLAS - Gevazp_cublas__ 74
Figura 4-13: Geração paralela de cenários em MPI _____ 75
Figura 4-14: Cluster de GPUs: geração paralela de cenários em MPI _____ 77
Figura 4-15: Utilização de *CUDA-Streams* na solução híbrida MPI e CUDA _____ 78
Figura 5-1: Árvore de cenários dos 12 PMOs _____ 79
Figura 5-2. Tomada de tempos para avaliação do desempenho _____ 80
Figura 5-3: Comparação dos tempos da versão serial e da Gevazp_gm na inicialização ___ 82
Figura 5-4: Detalhes dos tempos das versões *double* e *float* da Gevazp_gm na inicialização 82
Figura 5-5: Speedup obtidos nas versões *double* e *float* da Gevazp_gm na inicialização ___ 82
Figura 5-6: Comparação dos tempos da versão serial e da Gevazp_sh na inicialização ____ 83
Figura 5-7: Detalhes dos tempos das versões *double* e *float* da Gevazp_sh na inicialização 83
Figura 5-8: Speedup obtidos nas versões *double* e *float* da Gevazp_sh na inicialização____ 83
Figura 5-9: Comparação dos tempos da versão serial e da Gevazp_CUBLAS na inicialização _____ 84
Figura 5-10: Detalhes dos tempos das versões *double* e *float* da Gevazp_CUBLAS na inicialização_____ 84
Figura 5-11: Speedup obtidos nas versões *double* e *float* da Gevazp_CUBLAS na inicialização_____ 84
Figura 5-12: Speedup da primeira iteração da rotina OPTRA na versão Gevazp_sh _____ 86
Figura 5-13: Resultados dos tempos da rotina OPTRA na convergência _____ 87
Figura 5-14: Resultados de Speedup na convergência da rotina OPTRA _____ 88
Figura 5-15: Árvore com 6000 cenários_____ 90
Figura 5-16: Tempos de agregação e total - árvore com 6000 cenários_____ 92
Figura 5-17: Speedups de agregação e total - Árvores com 6000 cenários _____ 92

ÍNDICE DE TABELAS

Tabela 2-1. Qualificadores de função em CUDA_____ 21

Tabela 3-1. Capacidade Instalada no SIN [19]_____ 31

Tabela 3-2: Número de iterações e tempo total de execução da versão serial original (Versão 1) e da versão serial modificada (Versão 2) _____ 51

Tabela 5-1. Resultados obtidos na medida dos tempos de inicialização _____ 81

Tabela 5-2: Resultados obtidos dos speedups da etapa de inicialização _____ 81

Tabela 5-3: Resultados da agregação das séries - Gevazp_sh - float_____ 89

Tabela 5-4: Tempo total e overall speedup - Gevazp_sh – float _____ 89

Tabela 5-5: Tempo total e overall speedup - Gevazp_sh – float com 6000 aberturas _____ 90

1 INTRODUÇÃO

1.1 Considerações iniciais

A modelagem computacional é uma área multidisciplinar que realiza simulações numéricas para modelar um processo real ou um fenômeno natural. Através dessas simulações é possível testar teorias e avaliar diferentes cenários em áreas como engenharia, biologia molecular, dinâmica dos fluidos, fenômenos físicos e modelos de previsão. Tais simulações são complexas e podem levar a tempos computacionais elevados, podendo chegar a horas ou até mesmo dias.

Devido à crescente demanda por recursos computacionais, um dos maiores desafios da Ciência da Computação na atualidade é fornecer soluções computacionais capazes de viabilizar a redução de tempo mantendo-se a precisão dos resultados ou ainda proporcionando uma modelagem mais complexa executar no mesmo tempo computacional. A Lei de Moore [1] previa um crescimento dos transistores dos *chips*, dobrando a cada 18 meses, que era traduzido em ganhos da velocidade de *clock* na mesma proporção, e consequentemente do número de instruções executadas por segundo. Entretanto as dificuldades crescentes para se conseguir o aumento da frequência de *clock*, devido às limitações físicas, motivou os fabricantes a investirem nas tecnologias *multi-cores* e *many-cores*, impulsionando a área de Computação Paralela.

Computação Paralela é um paradigma que consiste em dividir um problema em partes menores e independentes entre si, de forma que possam ser executadas em diferentes unidades de processamento ao mesmo tempo, reduzindo o tempo total de execução. A Computação Paralela é uma das áreas dentro da Computação de Alto Desempenho. Uma forma relativamente barata de computação de alto desempenho e que se popularizou nos últimos anos, utiliza um agrupamento de computadores interligados por uma rede de alta velocidade, conhecido como *cluster* de computadores. O desenvolvimento de *softwares* paralelos em *clusters* utiliza bibliotecas de troca de mensagens entre os processadores do *cluster*, que viabilizam a troca de dados e sincronismo durante a execução do programa paralelo.

No trabalho [2], foi proposto a aplicação de processamento paralelo em *cluster* de computadores aplicado ao problema de planejamento da operação energética em sistemas

hidrotérmicos de geração de energia elétrica. Neste tipo de sistema, a geração hidroelétrica de baixo custo deve ser contrabalanceada com a geração térmica de custo mais elevado. O objetivo do planejamento da operação energética de sistemas hidrotérmicos é determinar, para cada estágio do horizonte de planejamento, metas para geração das usinas hidroelétricas e usinas termoelétricas de maneira a otimizar o valor esperado do custo futuro da operação, atendendo a determinados critérios de segurança. O autor aplicou técnicas otimizadas de processamento paralelo no modelo de planejamento de longo prazo NEWAVE [3], utilizando a biblioteca de troca de mensagens MPI [4], reduzindo de maneira eficiente o tempo de execução do modelo em *cluster* de computadores.

Dando continuidade à pesquisa desenvolvida em [2] e com o intuito de pesquisar possíveis melhorias nas soluções paralelas dos modelos computacionais que necessitem de grande capacidade de processamento, este trabalho tem como principal motivação investigar o poder de processamento das placas GPUs NVIDIA [5] no processamento paralelo, aplicado aos modelos de planejamento de geração hidrotérmica de energia. O alto poder de processamento das GPUs é consequência de sua arquitetura *many-cores*, onde uma grande quantidade de *threads* executam a mesma instrução em paralelo nos diversos *CUDA-cores*. As GPUs podem ter centenas ou até milhares de *CUDA-cores*, com desempenhos que chegam a Tera-flops, bem acima do poder de processamento das CPUs atuais.

Como a geração hidroelétrica está intimamente ligada às afluências futuras dos reservatórios, os modelos de planejamento da operação energética de longo e curto prazo utilizam como dados de entrada os cenários de afluências e energias gerados pelo modelo GEVAZP [6]. Dependendo de sua configuração, o modelo GEVAZP pode demandar muitos recursos computacionais, podendo chegar a horas de execução. Assim, a crescente utilização das GPUs na busca de desempenho de aplicações científicas paralelas, o alto poder de processamento das placas NVIDIA aliado a seu baixo custo em relação a um *cluster* de computadores e a necessidade de redução dos tempos computacionais dos modelos de planejamento da operação hidrotérmica, foram as principais motivações desta pesquisa.

O presente trabalho propõe a investigação de técnicas de processamento paralelo utilizando GPUs na redução do tempo computacional do modelo de geração de cenários sintéticos de afluências e energias – GEVAZP. O principal objetivo deste trabalho é compreender a arquitetura das placas GPUs, investigar o modelo de programação CUDA e pesquisar como desenvolver código otimizado em CUDA de maneira a extrair o máximo de

desempenho das GPUs. Desta maneira, pretende-se avaliar o poder de processamento das GPUs reduzindo-se os tempos de execução do modelo GEVAZP através desta solução paralela, utilizando a plataforma de programação CUDA. Além disso, este trabalho também tem como objetivo avaliar o desempenho do modelo GEVAZP em um *cluster* de computadores, reduzindo seu tempo através da utilização da biblioteca MPI.

Ao longo deste trabalho será apresentada uma metodologia detalhada para otimização da solução em CUDA, que é o objetivo principal desta dissertação. A implementação em MPI será avaliada em termos de desempenho, mas servirá também como um arcabouço para uma futura solução híbrida, MPI/CUDA, possibilitando a utilização futura em um *cluster* de GPUs.

1.2 Organização do trabalho

Além deste capítulo, o trabalho está dividido da seguinte forma:

O Capítulo 2 apresenta os principais conceitos de computação paralela utilizados neste trabalho e em seguida apresenta a arquitetura das GPUs NVIDIA e o modelo de programação CUDA.

O Capítulo 3 apresenta uma introdução ao problema de planejamento de geração hidrotérmica de energia elétrica, com ênfase no modelo de geração de cenários sintéticos de energia e afluências – GEVAZP. São apresentadas duas propostas de paralelização do modelo GEVAZP, utilizando-se CUDA em placas GPUs para acelerar o algoritmo de agregação das séries e MPI em *cluster* de computadores para geração das séries em paralelo.

O Capítulo 4 descreve em detalhes uma metodologia otimizada na implementação da paralelização do modelo GEVAZP em CUDA. Foram desenvolvidas duas versões paralelas explorando técnicas avançadas de acesso aos diferentes tipos de memória da GPU, mitigando os pontos que impactavam o desempenho computacional. Também foi desenvolvida uma versão utilizando a biblioteca paralela de álgebra linear CUBLAS [7]. Em seguida é apresentada a paralelização da geração das séries de afluências em *cluster* de computadores, utilizando a biblioteca de troca de mensagens MPI. Por fim é apresentada uma solução híbrida integrando as soluções em MPI e em CUDA.

No Capítulo 5 primeiramente são apresentados os resultados computacionais obtidos nas versões da estratégia de paralelização desenvolvida em CUDA. É feita uma análise dos resultados avaliando o desempenho computacional da metodologia proposta. Em seguida são apresentados os resultados da versão híbrida utilizando destacando os ganhos de desempenho da versão integrando MPI e CUDA.

Finalmente, o Capítulo 6 apresenta as conclusões deste trabalho assim como as propostas de trabalhos futuros, para continuidade desta pesquisa.

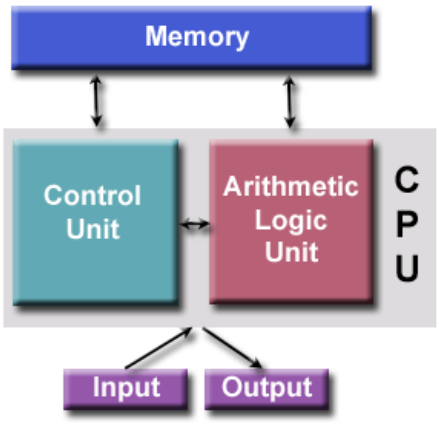
2 ARQUITETURA DAS GPUS NVIDIA E O MODELO DE PROGRAMAÇÃO CUDA

O objetivo deste capítulo é apresentar o conceito de GPU, iniciando com um breve histórico, mostrando em seguida as principais características da arquitetura das GPUs da NVIDIA e os conceitos essenciais do modelo de programação CUDA, elucidando como funciona e como a plataforma CUDA é usada na linguagem de programação C/C++. Antes porém, faz-se necessário apresentar alguns conceitos básicos de computação paralela abordados durante o desenvolvimento deste trabalho.

2.1 Computação Paralela

Computação paralela é o uso simultâneo de múltiplos recursos computacionais para reduzir o tempo de resolução de uma aplicação. O problema é quebrado em tarefas que possam ser resolvidas concorrentemente e, coordenadas por algum mecanismo, são processadas ao mesmo tempo e sincronizadas entre si, reduzindo o tempo total de execução. Computação paralela vem sendo amplamente utilizada no meio acadêmico e de pesquisa, para acelerar aplicações científicas de alta complexidade que demandam grandes recursos computacionais, como por exemplo, simulações atmosféricas, previsão do clima, modelos físicos, e também na engenharia.

John Von Newman, em 1945, foi o primeiro a propor um modelo de arquitetura para os computadores eletrônicos, que deveriam conter os seguintes componentes básicos [8]:



- **Memória** - acesso de leitura e escrita aleatório
- **Unidade de controle** – carrega as instruções e dados da memória
- **Unidade lógica e aritmética** – faz as operações aritméticas básicas
- **Entrada e saída** – interface com o homem

Figura 2-1: Arquitetura de computadores de Von Newman [8]

Baseado no modelo de Von Newman, a classificação das arquiteturas de computadores paralelos mais utilizada, foi proposta por Flynn em 1966, conhecida como Taxonomia de Flynn [9]:

- *SISD - Single Instruction Single Data.* Apenas um conjunto de instruções é executado por ciclo de *clock*, em apenas um conjunto de dados. Encaixam-se nessa classificação os computadores seriais de Von Newman.
- *SIMD – Single Instruction Multiple Data.* Todas as unidades de processamento executam o mesmo conjunto de instruções em diferentes conjuntos de dados. Nessa classificação encaixam-se os coprocessadores como Xeon Phi e as unidades de processamento gráfico, GPUs.
- *MISD- Multiple Instruction Single Data*. Nessa arquitetura, cada unidade de processamento executa uma instrução diferente para o mesmo dado.
- *MIMD- Multiple Instruction Multiple Data.* As unidades de processamento executam instruções independentes que operam em diferentes conjuntos de dados. Exemplos são as CPUs com *multi-cores* e *clusters* de computadores.

Dessas arquiteturas de computação paralela, duas são de especial interesse neste trabalho: (1) SIMD - atualmente utilizada nas placas de processamento gráfico, GPUs, que possuem um alto desempenho computacional. A arquitetura de memória das GPUs é de memória compartilhada, onde diferentes *threads* compartilham o mesmo endereço lógico de memória. (2) MIMD - utilizada nos *clusters* de computadores, onde diferentes computadores são interligados em uma rede de alta velocidade. A arquitetura de memória dos *clusters* é a de memória distribuída, onde os processadores do *cluster* trocam informações de dados e sincronismo através de troca de mensagens.

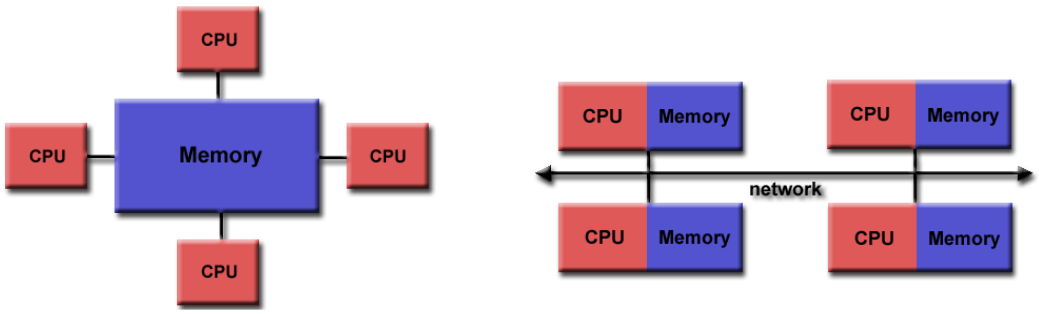


Figura 2-2: Arquiteturas de memória compartilhada, (à esq.) e distribuída (à dir.) [8]

Para melhor entendimento das próximas seções, “Arquitetura das GPUs NVIDIA” e “Modelo de programação CUDA”, faz-se necessário uma breve explicação dos conceitos e termos comuns mais utilizados em computação paralela [8]:

- ***High Performance Computing – HPC*** – Computação de alto desempenho se refere ao uso de *cluster* de computadores ou de supercomputadores para resolver tarefas que necessitam de grandes recursos computacionais em tempos menores do que levariam em um computador comum.
- **Nó** – identifica um único computador em um *cluster* de computadores. Pode ser composto por várias CPUs, processadores, memória e interface de rede. Os nós de um *cluster* são interligados para formar supercomputadores ou *clusters* de computadores.
- **CPU/core/processador** – CPU é a central de processamento dos computadores. Um computador pode ter várias CPUs. A CPU pode ser composta por vários processadores, também conhecidos como *cores*. Um nó com várias CPUs irá conter dezenas de cores.
- **Tarefa** – é um programa ou parte de um programa computacional com um conjunto de instruções que devem ser executadas pelos processadores das CPUs.
- ***Pipelining*** – consiste em subdividir uma tarefa em estágios de instruções encadeadas, onde cada estágio pode ser executado em paralelo com os demais, diminuindo o tempo final de processamento. Normalmente já implementado no *hardware* (CPUs e GPUs), onde tarefas como *load/store* dos dados de memória e processamento nas unidades lógicas e aritméticas são executadas em elementos independentes de hardware e podem ser executados em paralelo.
- **Memória compartilhada** – do ponto de vista do *hardware*, é uma arquitetura de computador em que todos os processadores têm acesso à mesma memória física. Do ponto de vista do modelo de programação, descreve um modelo onde tarefas paralelas acessam os mesmos endereços lógicos de memória, não importando sua localização física no *hardware*.
- ***SMP – Symmetric multi-processor*** – descreve a arquitetura de *hardware* de memória compartilhada onde os multiprocessadores compartilham um único espaço de endereço de memória e têm acesso igualmente a todos os recursos.
- **Memória Distribuída** – do ponto de vista do hardware é uma arquitetura de acesso à memória de computadores em uma rede, onde os computadores não têm

acesso à mesma memória física. Do ponto de vista de modelo de programação, descreve um modelo onde um conjunto de tarefas é executado em cada processador em sua memória local, e as tarefas são coordenadas através de troca de mensagens entre os processadores de uma mesma rede.

- **Comunicação** – trata-se da troca de informações entre as tarefas paralelas. Essas informações pode-se tratar de troca de dados entre os processadores ou ainda pontos de sincronismo entre as tarefas. Essa comunicação pode ser feita através de memória compartilhada ou enviar informações entre os processadores interligados em uma rede.
- **Sincronização** – é a coordenação em tempo real de tarefas paralelas, normalmente associadas à comunicação.
- **Granularidade** – Em computação paralela, granularidade diz repeito à razão entre o tempo de processamento e o tempo gasto em comunicação. Granularidade "grossa" quer dizer que grandes quantidades de processamento são feitas entre os pontos de comunicação. Granularidade "fina" quer dizer que pequenas quantidades de processamento são realizadas entre as comunicações.
- ***Speedup*** – ou ganho de velocidade, ou aceleração: é uma medida quantitativa do ganho de desempenho de códigos paralelos:

  Sp = Ts / Tp onde:

  Sp = Speedup alcançado

  Ts = Tempo de execução do algoritmo serial

  Tp = Tempo de execução do algoritmo paralelo

- **Eficiência** – é uma medida relativa ao grau de aproveitamento dos recursos computacionais. Mede a razão entre o desempenho paralelo obtido e os recursos disponíveis.

  Eficiência = (S_p / p) x 100% onde:

  Sp = Speedup alcançado para p processadores

  p = número de processadores

- ***Overhead* paralelo** – em computação paralela, é o tempo extra gasto para realizar transferência de dados entre processadores e nos pontos de sincronismo entre as tarefas paralelas.
- **Massivamente paralelo** – se refere a um grande número de processadores para realizar um conjunto de tarefas de forma coordenada.
- ***Embarassingly parallel*** – se refere a um tipo de problema onde pouco esforço é necessário para separar um problema em várias tarefas seriais. O problema já é paralelo por natureza.
- **Escalabilidade** – em termos de *hardware* se refere à capacidade de aumentar os recursos computacionais apenas adicionando novos elementos. Do ponto de vista de software, se refere à capacidade de uma solução paralela aumentar seu desempenho na mesma proporção (ou com pouca degradação de desempenho) em que mais recursos são utilizados.
- ***Threads*** – é um processo ou uma linha de execução que pode ser executado concorrentemente a outros processos. Nas CPUs, são chamadas *CPU-threads* ou apenas de *threads*. Nas GPUs, são conhecidas como *CUDA-threads.*
- ***TLP (Thread Level Parallelism)*** - paralelismo no nível de *threads* é uma forma de paralelizar um código serial distribuindo tarefas entre diferentes *threads* para serem executadas em processadores no mesmo computador, ou em vários *CUDA-cores*, na GPU.
- ***ILP (Instruction Level Parallelism)*** – paralelismo no nível de instrução é uma medida de quantas operações podem ser realizadas simultaneamente em um determinado processador/*core*. ILP é possível já que uma sequência de instruções necessita de unidades funcionais diferentes (unidades de *load/store*, ALUs, de cálculos de ponto flutuante - FP, multiplicadoras, etc.).
- **Latência** – é o tempo mínimo para enviar uma mensagem de um ponto a outro. Em termos de acesso à memória, é o tempo mínimo necessário para um processo/*thread* ler ou escrever em um determinado endereço de memória. Normalmente é medido em milisegundos (ms).
- **Largura de banda** – é a quantidade de dados que podem ser comunicados em uma unidade de tempo. Do ponto de vista da memória, é a taxa de dados que podem ser acessadas por segundo. Normalmente é medido em MB/s ou GB/s

2.2 Arquitetura das GPUs NVIDIA

Antes de apresentar os detalhes das GPUs NVIDIA duas perguntas devem ser respondidas: “O que é computação paralela em GPU?” e “Como as GPUs aceleram as aplicações?” Computação paralela em GPU é um modelo de computação heterogêneo que utiliza *Graphics Processing Units* (GPUs) junto com CPUs para acelerar aplicações científicas em diferentes áreas como engenharia, física, química, matemática, algoritmos, entre outros [10]. Sua arquitetura constituída por uma grande quantidade de pequenos núcleos de processamento, conhecida como *many-cores*, é adequada para executar em paralelo o mesmo conjunto de instruções em vários de seus *GPU-cores*. Enquanto as CPUs podem ser compostas por dezenas de *CPU-cores*, as GPUs oferecem um alto poder computacional através de milhares de *GPU-cores* que podem executar milhões de *threads* em paralelo (Figura 2-3).

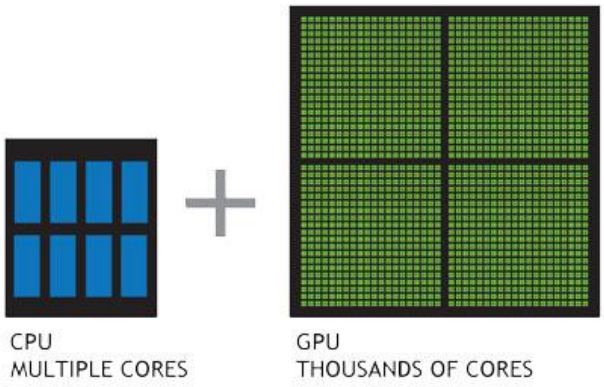


Figura 2-3: GPU tem milhares de cores [10]

Computação paralela baseada em GPUs acelera as aplicações executando as funções com pouca demanda de processamento na CPU, e as porções do código da aplicação de processamento intensivo em paralelo na GPU (Figura 2-4). Do ponto de vista do usuário, as aplicações simplesmente executam mais rápido [10].

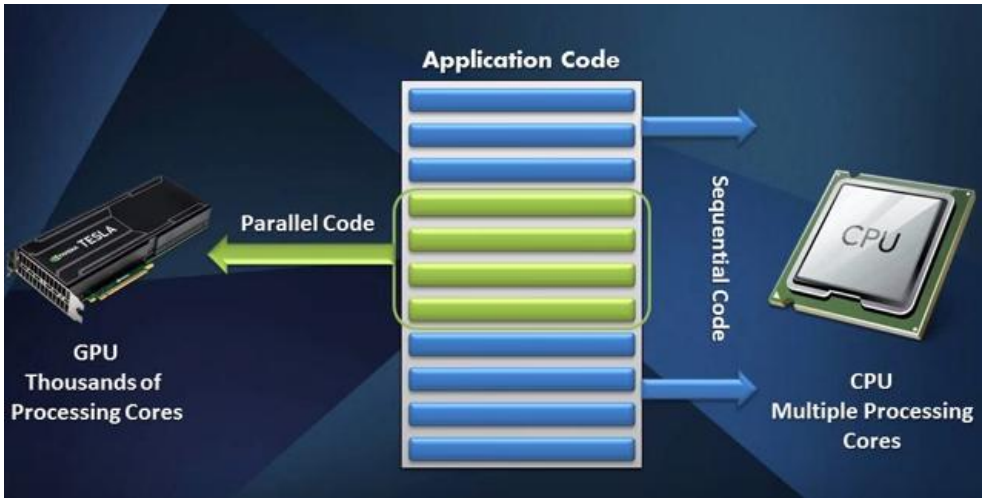


Figura 2-4: Como a GPU acelera as aplicações [11]

2.2.1 Graphics Processing Unit - GPU

Devido à crescente demanda pelo processamento de dados gráficos, principalmente processamento digital de imagens em *softwares* específicos e jogos, a empresa NVIDIA lançou sua primeira placa de vídeo para uso específico em 1999, conhecida como GPU. A capacidade de realizar cálculos em ponto flutuante, normalmente é medida em Gigaflops (10^9 *operações de ponto flutuante por segundo*), ou GFLOP/s. Nas GPUs o desempenho medido em GFLOP/s pode chegar a dezenas ou até centenas de vezes mais rápido que nas CPUs. A GPU consiste de processadores *many-cores*, que executam milhares de *threads* simultaneamente e possuem um enorme poder de processamento computacional aliado a uma grande largura de banda de acesso à memória, como pode ser visto na Figura 2-5 e Figura 2-6 (*single precision* utiliza 4 *bytes* e *double precision* utiliza 8 *bytes*).

Originalmente as GPUs foram projetadas para serem utilizadas em computação gráfica, para renderização de gráficos 2D e 3D, realizando cálculos em ponto flutuante e entregando essa informação para a saída de vídeo com uma alta taxa de dados. Seu elevado número de *stream processors* (SPs) é perfeitamente adequado para cargas de trabalho em dados de forma paralela, consistindo de milhares de *threads* independente executando pequenos programas (*shaders)* para cálculos de vértices, cálculos de transformação e rotação em formas geométricas, coloração de *pixels*, texturas entre outras técnicas utilizadas em computação gráfica.

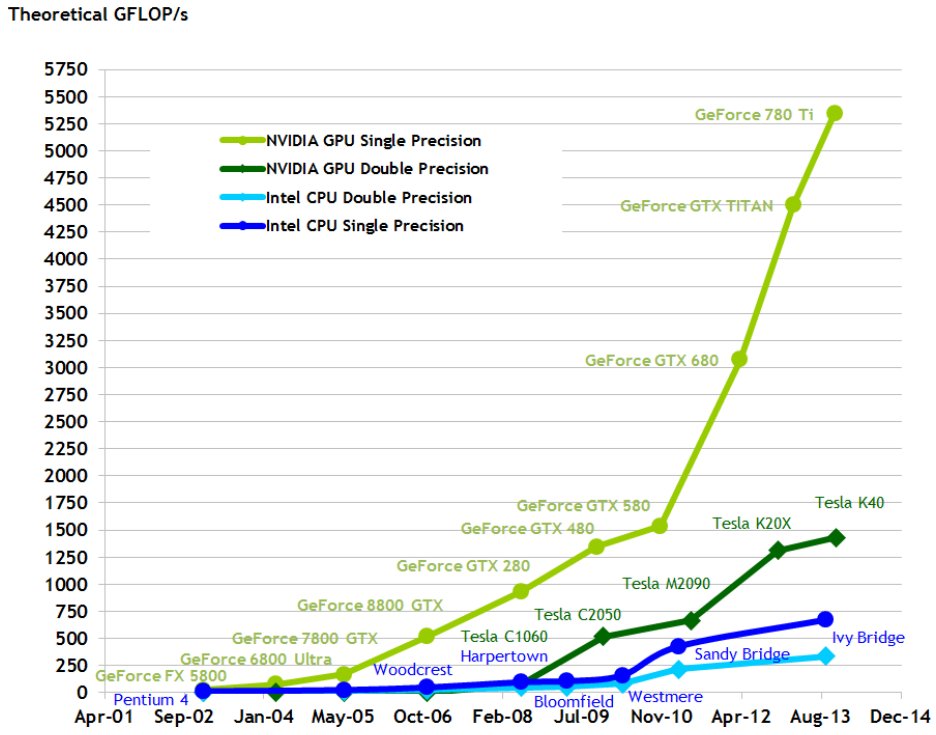


Figura 2-5: Operações de ponto flutuante por segundo em CPU e GPU (GFLOP/s) [5]

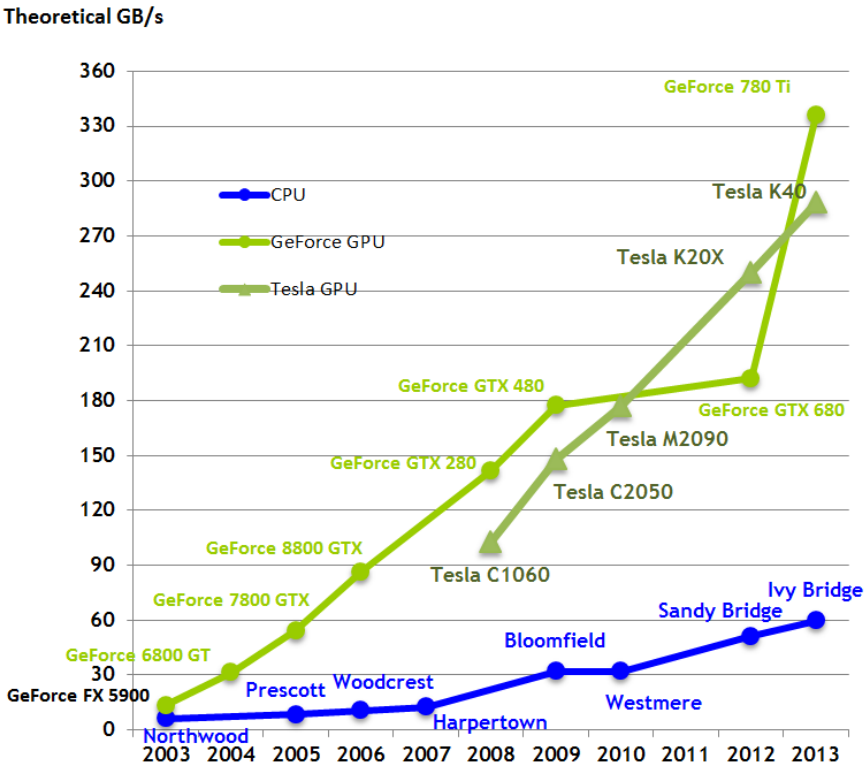


Figura 2-6: Largura de banda de acesso à memória para CPU e GPU [5]

O fato das aplicações gráficas serem extremamente paralelas reduz a necessidade de se manter dados em *cache* e realizar controles de fluxo extremamente complexos. Ou seja, enquanto na CPU boa parte de seus circuitos são destinados à memória *cache* e unidades de controle, a GPU possui mais unidades lógicas e aritméticas (ALUs) o que a torna muito mais eficiente em termos de processamento paralelo (Figura 2-7).

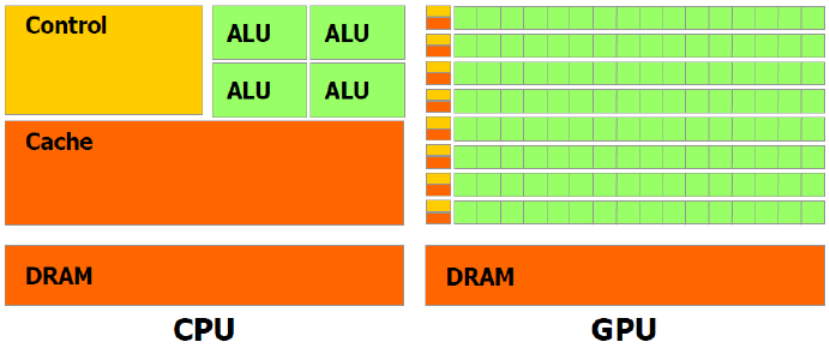


Figura 2-7: Comparação entre as arquiteturas da CPU e GPU [5]

Vislumbrando o uso desse tipo de dispositivo para outras aplicações que poderiam se beneficiar de sua capacidade de processamento, em 2006 a NVIDIA lançou sua placa de vídeo GeForce 8 *series*, sua primeira GP-GPU (*General Purpose Computing on Graphics Processing Unit*), um marco para o próximo passo em termos de evolução em GPUs: GPU vista como um conjunto de processadores massivamente paralelos, totalmente programáveis. A arquitetura da placa G80 (GeForce 8800) foi a primeira a ter *shaders* programáveis de forma unificada, ou seja, um processador unificado completamente programável chamado de *Stream Multiprocessor*, ou simplesmente SM. Com esse novo projeto de *hardware* as GP-GPUs se tornaram dispositivos padronizados, escaláveis e com alto poder de processamento, disponíveis para os desenvolvedores de *software* paralelo para computação de alto desempenho, podendo ser utilizado tanto em *softwares* comerciais quanto em computação científica em grandes centros de pesquisas de diferentes áreas.

Para tornar viável a programação para GP-GPU (ou apenas GPU), a NVIDIA lançou em Fevereiro de 2007 uma plataforma de programação paralela chamada CUDA (*Compute Unified Device Architecture*). Sua arquitetura é baseada em execução de ***threads*** em cada um dos *CUDA-cores* da GPU. Ao se realizar uma chamada de função na GPU, esta é capaz de criar, gerenciar, agendar e executar as *threads*, que atuarão em um conjunto separado de dados, de acordo com o modelo SIMD, coordenadas pelo *Stream Multiprocessor*.

2.2.2 Stream Multiprocessor e paradigma SIMT

A arquitetura das GPUs NVIDIA foi construída como um conjunto escalável de *Stream Multiprocessors*. Cada SM é composto por 8, 32, 48 ou 192 *CUDA-cores* denominados de *stream processors (SP).* Cada *CUDA-core* possui *pipelines* de operações aritméticas (*ALUs – Arithmetic Logic Unit*) e de operações de ponto flutuante (*FPU – Floating Point Unit*). Compartilham recursos comuns como *cache* L1, memória local, registradores, unidades de *load/store* e escalonadores de threads, tendo também acesso à memória global da GPU.

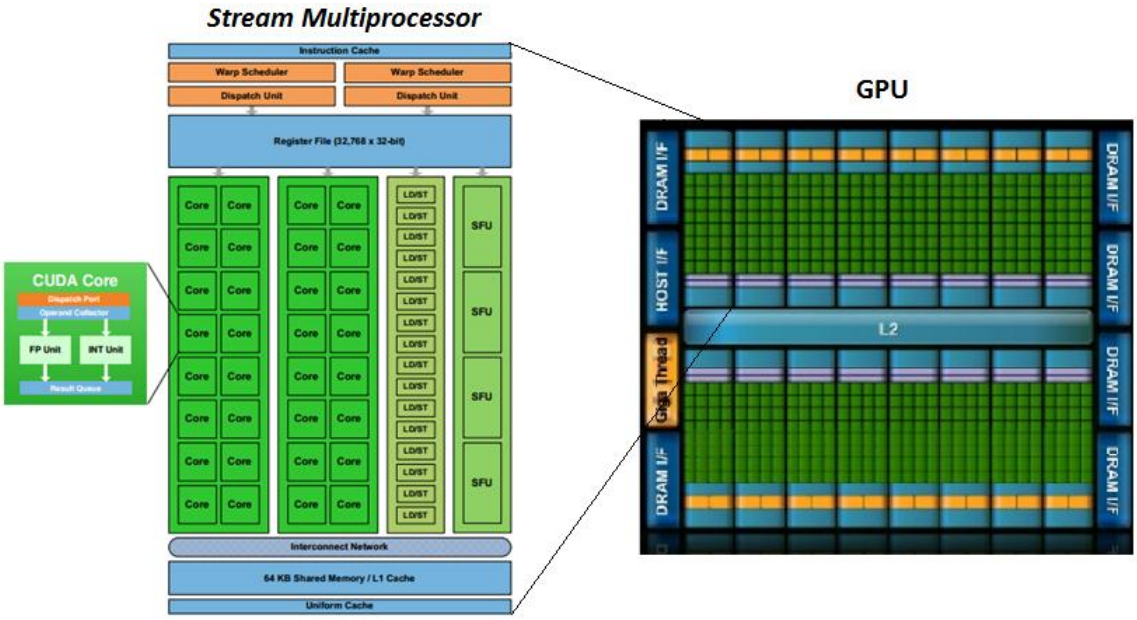


Figura 2-8: Detalhes dos CUDA-cores, do Stream Multiprocessor e da GPU [5]

Quando um programa escrito em CUDA executa um conjunto de instruções na GPU, os SMs gerenciam a criação e execução concorrente de *threads* em seus processadores. Cada *thread* alocada em um SM, executa um código na GPU conhecido como *kernel*, e todas as *threads* dentro de um bloco alocado nesse SM executam o mesmo conjunto de instruções. Como sua arquitetura é completamente escalável, a GPU pode ser composta por mais de um *Stream Multiprocessor*. Cada SM é independente dos demais em termos de execução, ou seja, um SM pode lançar um bloco de *threads* executando um determinado conjunto de instruções enquanto outro SM pode executar outro conjunto de instruções completamente diferentes,

introduzindo um novo conceito de programação paralela conhecido como SIMT (*Single-Instruction Multiple-Thread*).

2.2.3 Warps

As *threads* executadas em um *Stream Multiprocessor* são organizadas em grupos de 32 *threads*, chamadas ***warps***. O SM cria, gerencia, agenda e executa as *threads* alocadas no *warp*. Dentro do SM, os *CUDA-cores* estão divididos em blocos de execução que possuem unidades independentes chamadas de *warp scheduler* e *instruction dispatch unit* [12]. Estas duas unidades gerenciam as *threads* e os *warps* para manter os *CUDA-cores* ocupados o máximo de tempo possível. Quando é dado a um SM um ou mais blocos de *threads* para execução, ele particiona em *warps* e entrega ao *warp scheduler* para execução. Para obter *multithreading* massivamente paralelo, os SMs têm que executar eficientemente operações de grande latência, como por exemplo, os acessos à memória global. Se uma determinada instrução sendo executada por uma *thread* precisa ficar esperando operações de alta latência, o *warp scheduler* seleciona um *warp* no qual as *threads* já estão prontas para executar na próxima instrução, realizando um *pipeline* de *threads* "ativas", ou seja, *threads* sendo executadas em um SP.

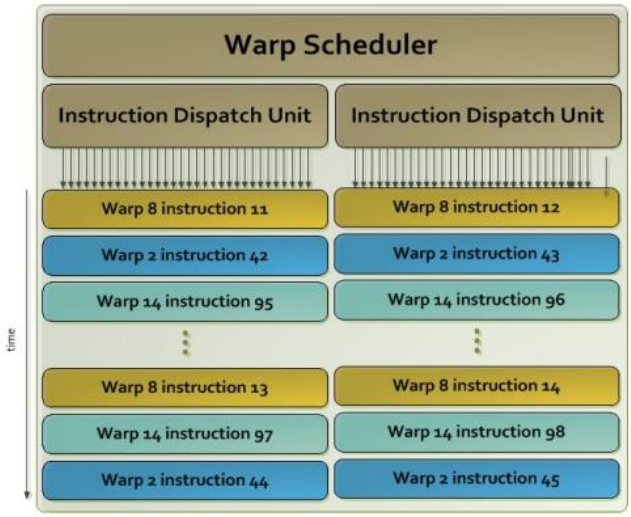


Figura 2-9: Cada SMx Kepler possui 4 *warp-scheduler* e 8 *instruction dispatch unit* [12]

Threads individuais que compõem um *warp* iniciam junto num mesmo endereço de programa, mas têm seus próprios endereços de instrução e registradores de estado, sendo dessa forma livres para executar independentemente. Como os *CUDA-cores* foram projetados para trabalhar em paralelo executando 32 instruções simultâneas, assim o máximo de

eficiência é obtido quando todas as 32 *threads* no *warp* percorrem o mesmo caminho de instruções. Caso as *threads* percorram caminhos diferentes devido a condições lógicas, como *if* e *switch* por exemplo, o *warp* serializa essas execuções, degradando assim o desempenho da aplicação [5]. Este é um ponto muito importante a ser observado pelo desenvolvedor CUDA, pois se deve evitar divergências nos códigos programados para GPU, buscando maximizar o desempenho da aplicação. Códigos com muitos ramos devidos a lógicas condicionais geralmente não são bons candidatos a serem portados para GPU.

A implementação em *hardware* do conceito de *multithreading* na GPU difere em vários aspectos em relação à CPU. Enquanto na CPU o contexto de execução (registradores, contador de programa, etc.) de cada *thread* executada pelo processador é mantido na memória RAM (*off-chip*), na GPU o contexto de cada *thread* sendo executada em um SM é mantida *on-chip, nos registradores,* durante toda a duração do *warp* a qual ela pertence. Assim a troca de um contexto de execução para outro tem custo praticamente igual a zero na GPU. A GPU realiza tanto paralelismo espacial, por exemplo, 192 *CUDA-cores* executando simultaneamente, quanto temporal (*multithreading*).

Em resumo, *multithreading* em GPU possui um baixíssimo custo para criação e gerenciamento das *threads* alocadas nos SMs, permitindo que a aplicação faça uso de milhares ou até milhões de *threads* simultâneas.

2.2.4 Categorias das GPUs NVIDIA

2.2.4.1 Família GeForce

As placas GeForce são otimizadas para aplicações em jogos. No momento em que se escreve esta dissertação, a placa mais atual desta família é a GeForce GTX Titan Z, otimizada para utilização em games 3D em resolução 4K. Construída sob a arquitetura Kepler [12], é equipada com 5.760 núcleos de processamento de 876 MHz divididos em 30 *Stream Multiprocessors*. Possui 12 GB de memória RAM com largura de banda de 672 GB/s e desempenho de 8,1 TFLOPS em precisão simples e 190 GFLOPS em precisão dupla.

2.2.4.2 Família Quadro

As placas Quadro são orientadas para estações de trabalho de visualização com *drivers* e *firmwares* desenvolvidos para aplicativos CAD (*Computer Aided Design*) e DCC (*Digital Content Creation*), com maior estabilidade (*clocks* mais baixos) e memórias maiores e com menos chances de gerar erros nos resultados, utilizando ECC (*Error Correction Code*). Têm preço maior que as GeForce que possuem o mesmo número de núcleos. A placa mais atual desta família é a NVIDIA Quadro M6000 com 3.072 núcleos de processamento de 1114 MHz divididos em 24 *Stream Multiprocessors*. Possui 12 GB de memória RAM com largura de banda de 317 GB/s e desempenho de 6,1 TFLOPS em precisão simples e 1,7 TFLOPS em precisão dupla. Enquanto a família GeForce fornece uma alta taxa de atualização de saída de vídeo devido à demanda por velocidade, a família Quadro é orientada para otimizar os resultados em texturas, sombreamento e processamento digital de imagens 3D com alto nível de detalhes. Estas placas são projetadas para utilização em animações 3D e edições de áudio e vídeo.

2.2.4.3 Família Tesla

GPUs da família Tesla são processadores gráficos especificamente projetados para computação de alto desempenho que oferecem um alto poder de processamento e uma memória dedicada maior. As GPUs desta família não têm uma conexão direta para dispositivos de exibição de vídeo, ou seja, não são orientadas a aplicações gráficas e sim a aplicações científicas e de empresas que necessitam de HPC para obter ganhos de desempenho em termos do tempo de execução. A placa mais atual dessa família é a Tesla K80 com 4.492 núcleos de processamento a 875 MHz e 24 GB de memória com largura de memória de 480 GB/s. Possui proteção ECC e um maior número de unidades de precisão dupla, fornecendo 8,7 TFLOPS em precisão simples e 2.9 TFLOPS em precisão dupla.

Em termos de evolução das GPUs da família Tesla, a arquitetura *Fermi* foi lançada em 2010 trazendo suporte para novas instruções em C++ como tratamento de exceções e alocação dinâmica de memória, além das unidades de *load and store* e as SFUs (*special functions)* que realizam cálculos de funções como seno, cosseno, raiz quadrada, etc.

A arquitetura Kepler [12], foi lançada em 2012 trazendo um novo modelo de SM, com 192 CUDA *cores*, chamado de SMx. Ela possui quatro *warp schedulers* e oito *instructions dispacths*, e o número de unidades de precisão dupla foi aumentado para 64 em cada SMx já que essas unidades são largamente utilizadas em computação de alto desempenho, além de 32 unidades de *load/store* e 32 unidades de SFUs (Figura 2-2-10).

Figura 2-2-10: SMx - 192 CUDA-cores de precisão simples e 64 de precisão dupla [12]

O grande avanço dessa arquitetura foi a introdução de dois novos conceitos com o objetivo de permitir aos desenvolvedores extrair o máximo de utilização da capacidade de processamento: Paralelismo Dinâmico e Hyper-Q.

O paralelismo dinâmico simplifica a programação da GPU permitindo aos programadores acelerar facilmente todos os enlaces paralelos aninhados. Permite que as *threads* executadas na GPU possam criar e executar outras *threads* com cargas de trabalho independentes, sem a necessidade de retornar os dados intermediários para a CPU, como nas arquiteturas anteriores, diminuindo o *overhead* de comunicação entre GPU e CPU (Figura 2-11).

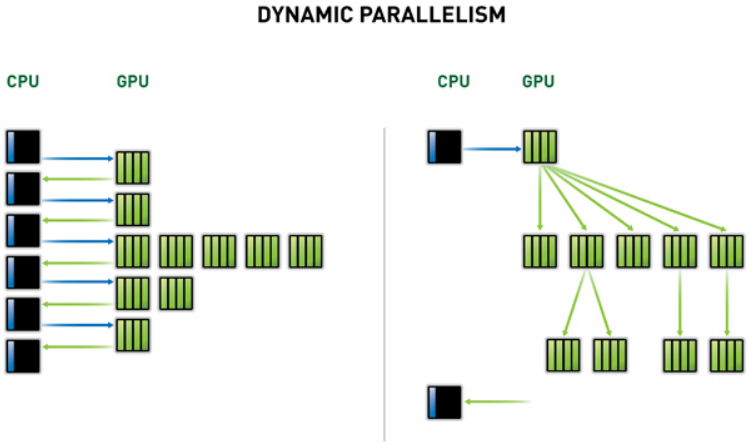


Figura 2-11: Paralelismo dinâmico [12]

O Hyper-Q diminui o tempo ocioso da CPU ao permitir que múltiplos núcleos de CPU utilizem simultaneamente uma única GPU Kepler. Esta funcionalidade aumenta o número total de conexões entre o *host* e o *device* para 32 conexões simultâneas, permitindo a interação entre processos MPI e *streams* CUDA ou ainda entre *threads* na CPU e CUDA *threads* na GPU, aumentando significativamente o uso dos recursos.

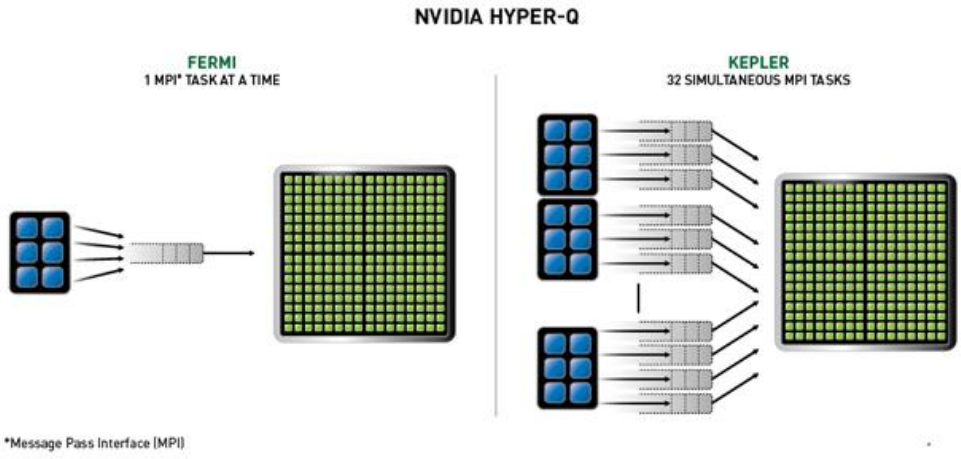


Figura 2-12: Hyper-Q permite que até 32 *streams* de diferentes processadores [12]

2.3 Modelo de Programação CUDA

CUDA (*Compute Unified Device Architecture*) é uma plataforma de computação paralela e um modelo de programação criados pela NVIDIA. Ela permite aumentos significativos de desempenho computacional ao aproveitar a potência da unidade de processamento gráfico, as GPUs [3]. Um modelo de programação é uma abstração da arquitetura do computador que atua como uma ponte entre uma aplicação e sua implementação em *hardware*. Ele fornece uma visão lógica para uma arquitetura específica. O programa escrito para um modelo de programação define como os componentes da aplicação compartilham informação e coordena suas atividades. O compilador é o componente responsável por mapear a estrutura lógica programada em instruções para o *hardware* específico. Esta comunicação é realizada através do compilador e de bibliotecas que utilizam primitivas de *hardware* e do sistema operacional [13].

A NVIDIA lançou em 2007 um guia de programação para a linguagem CUDA, que é uma extensão da linguagem C/C++ tornando a plataforma aceita por vários desenvolvedores de aplicações paralelas. Para desenvolver em linguagem CUDA deve-se primeiro fazer o *download* e instalar o *NVIDIA CUDA Toolkit* [14]. Ele fornece um ambiente de *software* que permite aos desenvolvedores usar sua extensão para C/C++ como uma programação de alto nível e construir suas aplicações paralelas aceleradas na GPU. O *CUDA Toolkit* inclui o compilador NVIDIA para GPUs, bibliotecas matemáticas e ferramentas para depuração de código e otimização do desempenho de suas aplicações. Além disso, disponibiliza também manuais de usuários, referências para a API e código fonte contendo exemplos de uso.

O modelo de programação CUDA fornece duas funcionalidades vitais para aproveitar o poder da arquitetura das GPUs:

- Uma maneira para organizar as *threads* através de uma estrutura hierárquica;
- Uma maneira de acessar a memória da GPU através de uma estrutura hierárquica;

O desenvolvimento em CUDA é baseado no modelo de programação descrito a seguir.

2.3.1 Kernels

A terminologia utilizada em CUDA define como ***host*** a parte da CPU e como ***device*** a parte da GPU. Em CUDA, uma função a ser executada no *device* é chamada ***kernel***. O código paralelo é escrito em uma função *kernel*, que quando chamada, será executada N vezes em paralelo por N *CUDA-threads*.

Uma função kernel é construída utilizando-se a palavra reservada **__global__** na definição da função. O qualificador **__global__** informa que esta função será chamada no *host* e executada no *device*. O número de *threads* que executam este *kernel* é definido no *host* e passado na chamada do *kernel*. Sua chamada é especificada usando uma nova sintaxe de execução <<< blocos, *threads* >>> logo após o nome do *kernel*. A Figura 2-13 ilustra a definição e execução de um *kernel*:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Figura 2-13: Definição e execução de um kernel

Além de **__global__**, a linguagem CUDA introduz mais dois novos qualificadores para as funções: **__host__** e **__device__**. O qualificador **__host__** informa ao compilador **nvcc** (NVIDIA) que esta função será chamada e executada apenas no *host*. Já o qualificador **__device__** permite que as funções sejam chamadas apenas e executadas apenas na GPU. A tabela abaixo resume o comportamento para todos os casos:

Tabela 2-1. Qualificadores de função em CUDA

Modificador	Utilização pela CPU e GPU
__host__	Função chamada no ***host*** e executada no ***host***
__global__	Função chamada no ***host*** e executada no ***device***
__device__	Função chamada no ***device*** e executada no ***device***

Obs: A evolução das GPUs e as funcionalidades introduzidas em cada versão da plataforma CUDA são especificadas através de um conceito chamado de *Compute Capability*. A partir da *Compute Capability* 3.0, as funções **__global__** podem ser chamadas do dispositivo também.

O fluxo de processamento entre CPU e GPU é composto basicamente por 3 passos:

- Cópia de dados do *host* para o *device*;
- Execução do *kernel* no *device*;
- Cópia de dados do device para o *host.*

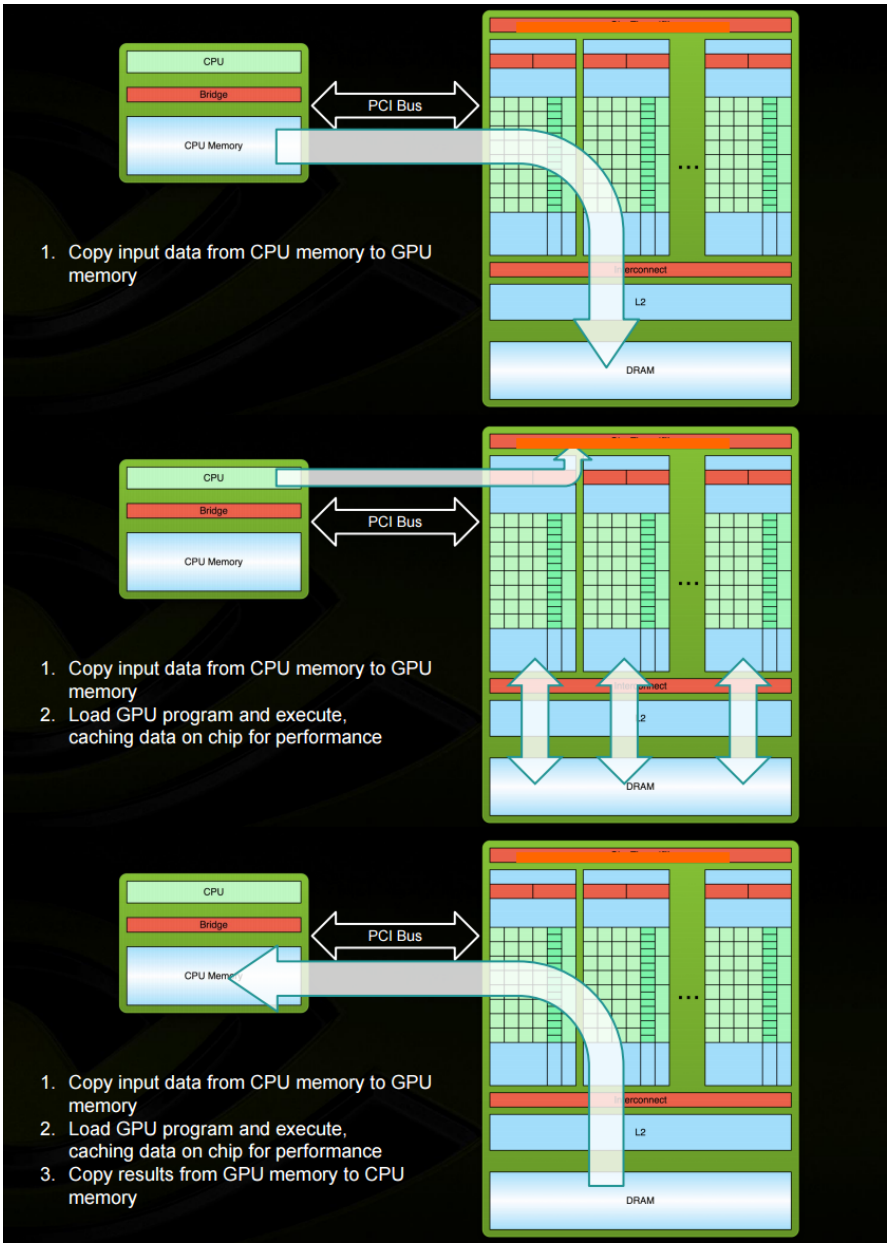


Figura 2-14: Fluxo de execução de um kernel

As funções *kernel* em C/C++ possuem algumas restrições de execução [13]:

- Podem acessar apenas a memória do *device*;
- Devem retornar o tipo *void*;
- Não podem ser recursivas;
- Não podem possuir variáveis estáticas;
- Não suportam ponteiros para função;
- Não suportam um número variável de argumentos;
- Possuem um comportamento assíncrono.

2.3.2 Hierarquia de *Threads*

Quando uma função *kernel* é chamada no *host*, a execução é transferida para o *device*, onde um elevado número de *threads* é criado e cada *thread* executa as instruções definidas no *kernel*. Saber como as *threads* são organizadas é fundamental para desenvolver código em CUDA. O escalonamento de *threads* na plataforma CUDA é feito por uma hierarquia de dois níveis, introduzindo dois novos conceitos: **bloco de *threads*** e ***grid* de blocos** (Figura 2-15). É com esses conceitos que se organiza a repartição dos dados entre as *threads*, bem como sua organização e distribuição no *hardware*.

2.3.2.1 Blocos de threads

Um bloco é a unidade básica de organização das *threads* e de mapeamento para o *hardware*. Um bloco de *threads* é alocado em um SM, e pode ter até três dimensões, x, y e z. As *threads* pertencentes a um mesmo bloco podem cooperar entre si, através de instruções de sincronismo entre as *threads* e acessar os mesmo endereços da memória compartilhada.

2.3.2.2 *Grid* de blocos

O *grid* é a unidade básica onde estão distribuídos os blocos. O *grid* é a estrutura completa de distribuição das *threads* que executam um *kernel*. É no *grid* que está definido o

número total de blocos e de *threads* que serão criados e gerenciados pela GPU para uma dada função. Um *grid* pode ter até 3 dimensões x, y e z.

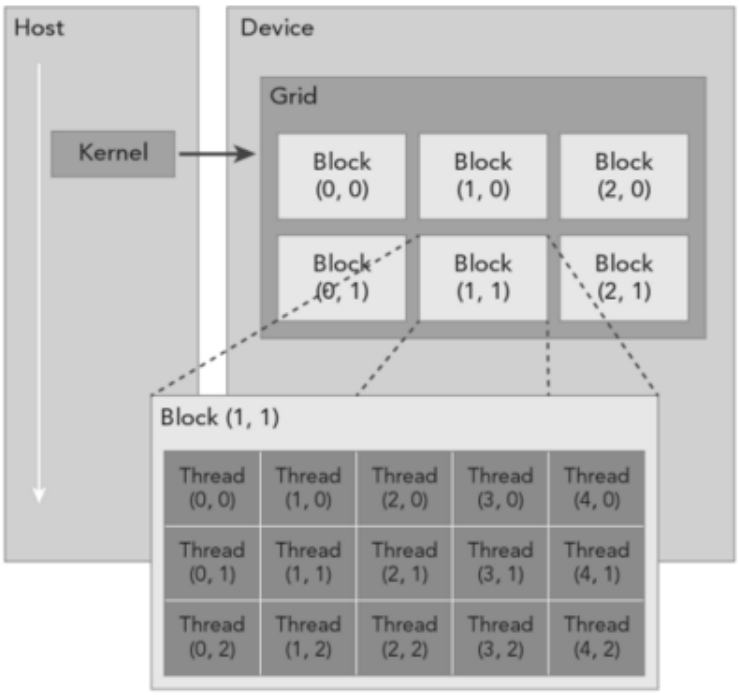


Figura 2-15: Organização das *threads* em Grids e Blocos

Fisicamente, este modelo é mapeado no *hardware* da seguinte forma (Figura 2-16):

- Cada ***thread*** é executada em um ***Stream Processor* (*SP*);**
- Cada **bloco** de *threads* é executado em um ***Stream Multiprocessor(SM)****;*
- Cada **grid** é executado por pelo menos um ***device (GPU)*.**

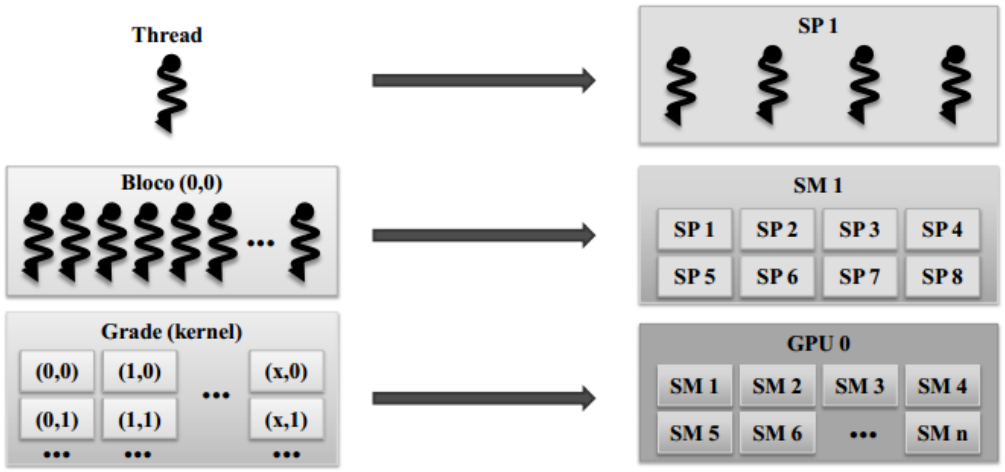


Figura 2-16: Mapeamento das threads para os processadores CUDA [15]

O número de *threads* por bloco e de blocos por *grid* é especificado na chamada do *kernel*, na sintaxe **<<<numBlocos, numThreads>>>**.

A identificação das *threads* é então baseada nesses conceitos. Cada *thread* tem seu ***id***, que a identifica unicamente na execução do *kernel*. Para a identificação dos blocos e *threads*, a plataforma fornece variáveis especiais conhecidas como *built-in variables*. Elas só são válidas dentro de funções executadas no *device*. CUDA fornece também um tipo especial para definição das dimensões x, y e z: **dim3** (x, y, z).

- **gridDim**: variável do tipo **dim3** que contém as dimensões do *grid*.
- **blockDim**: variável do tipo **dim3** que contém as dimensões do bloco.
- **blockIdx**: variável do tipo **uint3** que contém o índice do bloco no *grid*, das dimensões x, y e z.
- **threadIdx**: variável do tipo **uint3** que contém o índice do bloco no *grid*, das dimensões x, y e z.

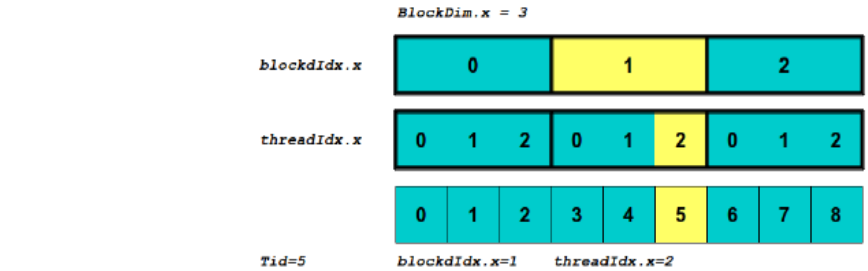
Através dessas variáveis é possível determinar um identificador único para cada *thread*, através da sua combinação da seguinte forma:

$$idThread = blockIdx.x * blockDim.x + threadId.x$$

Como *threadIdx* é um componente de 3 dimensões, pode-se identificar as *threads* usando índices de *threads* unidimensionais, bidimensionais ou tridimensionais, formando blocos de *threads* de até 3 dimensões. Essa característica fornece uma maneira natural de modelar o código paralelo em diferentes domínios, tal como vetores, matrizes e volumes.

Existe um limite para o número de *threads* por bloco, já que as *threads* de um bloco são alocadas em um mesmo SM e devem compartilhar os recursos. Atualmente o número máximo de *threads* em um bloco é 1024.

A figura abaixo ilustra a chamada de uma função *kernel* **add** que soma 2 vetores e o uso do threadId. No host a função é chamada colocando <<<>>> com as seguintes informações: ***add<<< NroBlocos, NroThreadsPorBloco>>>***



add_gpu_grid.cu

```
__global__ void add( int N, float *a, float *b, float *c ) {
    int tid = blockIdx.x*blockDim.x + threadIdx.x;    // this thread handles the data at its
          thread id
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

add_gpu_grid.cu

```
add<<< (N + blocksize-1)/blocksize, blocksize >>>( N, dev_a, dev_b, dev_c );
```

Figura 2-17: *Kernel* que soma dois vetores na GPU

2.3.3 Hierarquia de acesso à memória

As *threads* podem acessar múltiplos espaços de memória durante sua execução. Cada *thread* possui seu espaço privado de memória local. *Threads* pertencentes a um mesmo bloco podem cooperar entre si e compartilham o mesmo espaço de memória compartilhada (*Shared Memory*). As *threads* de um *grid* possuem uma visão geral do mesmo espaço de memória global (*Global Memory*), e ainda podem acessar ainda duas outros tipos de memória: memória de textura (*Texture Memory)* e memória constante (*Constant Memory*), que são tipos especiais de memória com tipos de acesso para aplicações específicas.

A Figura 2-18 ilustra o modelo lógico da hierarquia de acesso à memória descrita anteriormente. A Figura 2-19 apresenta o acesso em *hardware* para os diferentes tipos de memória da GPU.

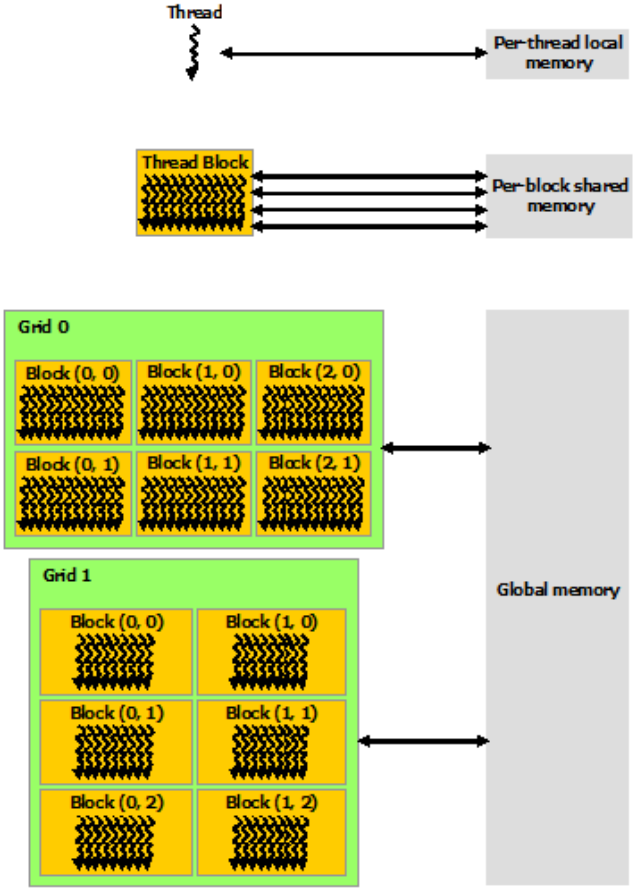


Figura 2-18: Hierarquia de Memória do modelo de programação CUDA [5]

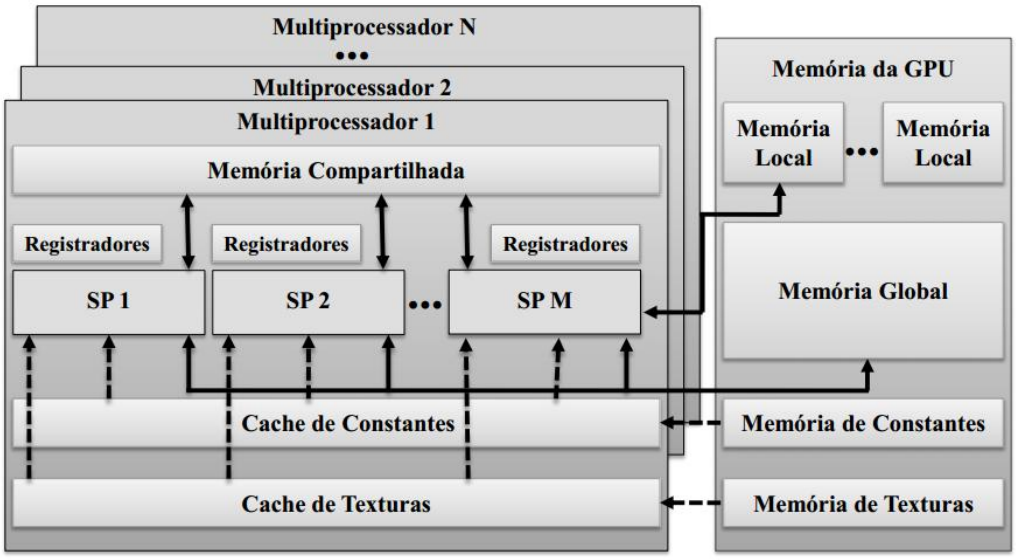


Figura 2-19: Acesso à Memória - Hardware da GPU [15]

A memória local dependendo do seu tamanho pode ser mapeado em registradores ou na memória global. Os registradores são o tipo de memória com acesso mais rápido, seguida

pela memória compartilhada. Sua latência é de apenas poucos ciclos. O acesso à memória global tem uma latência muito maior da ordem de centenas de ciclos. As memórias de textura e constante possuem um tempo de acesso que pode chegar à mesma ordem de grandeza da memória global.

O correto dimensionamento da memória local influencia muito no desempenho da aplicação, já que a latência de acesso à memória global é centenas de vezes maior do que nos registradores.

2.3.4 Modelo de escalabilidade da GPU

Quando um *kernel* é chamado para executar na GPU, os blocos do *grid* são entregues aos SMs para execução, e as *threads* pertencentes aos blocos executam em paralelo a sequência de instruções do SM em que foram alocadas. Ao terminar a execução, novos blocos são atribuídos aos SMs até o fim do processamento do *kernel*. Esse processo é independente do número de SMs e do número de CUDA-cores da GPU, ou seja, independente do *hardware*. Esse modelo de grid e blocos de *threads* facilita o mapeamento do código CUDA para o *hardware* na GPU, tornando a execução do *kernel* escalável de acordo com o número de SMs de dispositivo, e sem a necessidade de recompilar o código para diferentes modelos de GPU.

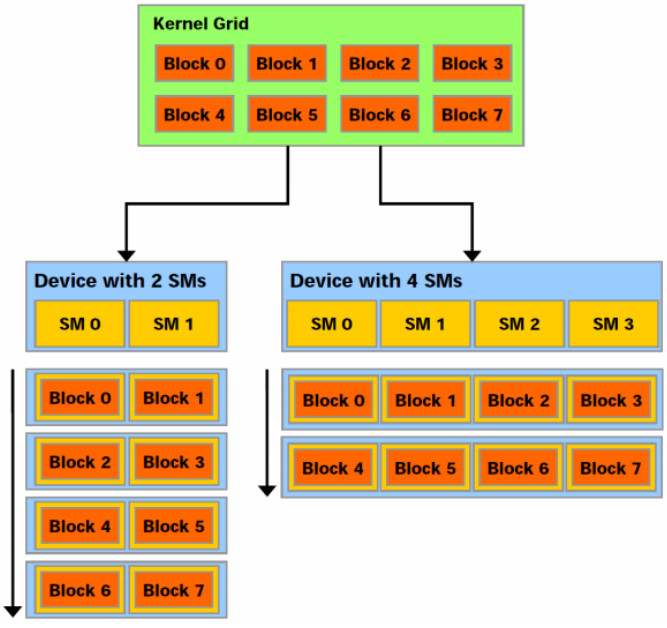


Figura 2-20: Execução de 8 blocos em GPUs com 2 SMs e 4 SMs [5]

2.3.5 Nível de ocupação do *Stream Multiprocessor*

Um elevado número de *threads* por SM permite executar aplicações que fornecem um alto grau do nível de ocupação do SM, também chamado de ocupância. Ocupância é definido como a razão entre o número de *warps* "ativos" em um *Stream Multiprocessor* e o número máximo de *warps* ativos suportados pelo SM [16]. Aumentando o nível de ocupação, pode-se esconder a latência de acesso a memória contribuindo para um melhor desempenho da aplicação.

O NVIDIA CUDA Toolkit [14] fornece um documento Excel, *"Occupancy Calculator"*, que permite calcular qual o tamanho de bloco e a quantidade de blocos por *Stream Multiprocessor* que maximizam a ocupância para cada GPU. Esta ferramenta é extremamente útil como uma ferramenta de aprendizado. Pode-se visualizar o impacto de como as mudanças nos parâmetros da execução (tamanho de bloco, número de registradores por *thread* e memória compartilhada por *thread*) afetam a ocupância, e consequentemente o desempenho da aplicação paralela.

3 PARALELIZAÇÃO DO MODELO DE GERAÇÃO DE CENÁRIOS SINTÉTICOS - GEVAZP

Este capítulo consiste de uma breve descrição do planejamento da operação de sistemas hidrotérmicos para geração de energia elétrica e da cadeia de modelos utilizados no planejamento da operação energética do sistema brasileiro. Além disso, são apresentados os principais aspectos teóricos do modelo de Geração de Cenários Sintéticos de Vazões e Energias – GEVAZP, utilizado nos modelos de planejamento da operação de longo, médio e curto prazo NEWAVE, SUISHI e DECOMP, desenvolvidos pelo CEPEL. Em seguida, é realizado um estudo das possíveis estratégias de paralelização do modelo GEVAZP, onde tais estratégias são definidas de acordo com a granularidade dos trechos paralelizáveis, apontando para cada caso as soluções de memória distribuída ou memória compartilhada de acordo com a natureza do problema. Por fim são propostas ao final do capítulo duas soluções para paralelização do modelo GEVAZP utilizando MPI na primeira e CUDA na segunda proposta. É proposta ainda uma solução híbrida coordenando as duas metodologias em MPI e CUDA para trabalharem juntas em um *cluster* de GPUs.

Estas soluções serão descritas e avaliadas nos capítulos subsequentes.

3.1 Planejamento da Operação de Sistemas Hidrotérmicos

Sistemas hidrotérmicos de geração de energia elétrica são compostos por usinas hidroelétricas e usinas termoelétricas. Neste tipo de sistema parte da energia é gerada com baixo custo através do volume de água armazenado nos reservatórios das usinas hidroelétricas contrabalanceando o custo mais elevado de geração das usinas termoelétricas [17] [18] [19]. O sistema hidrotérmico brasileiro é um sistema de grande porte, predominantemente hidráulico e com múltiplos proprietários. Tais características permitem considerá-lo único em âmbito mundial.

Tabela 3-1. Capacidade Instalada no SIN [20]

	MW	%
Hidro Nacional	72.283,4	72,13%
Hidro Itaipu	7.000,0	6,98%
Pequenas Hidros	686,5	0,69%
Térmica convencional	16.847,0	16,81%
Termonuclear	1.990,0	1,99%
Eólica	1.170,9	1,17%
Biomassa	239,2	0,24%
Total	100.217,0	100,00%

O SIN, Sistema Interligado Nacional, é formado pelas empresas da região Sul, Sudeste, Centro-Oeste, Nordeste e parte da região Norte. Apenas 1,7 % da energia requerida pelo país é produzida fora do SIN, em pequenos sistemas isolados [21].

No Brasil, cerca de 80 % do potencial de geração de energia é composto por usinas hidroelétricas (Tabela 3-1), aproveitando o grande potencial de energia fluvial, devido às características de relevo favoráveis somado ao grande potencial hídrico do nosso país [20]. A energia de uma usina hidroelétrica possui um custo direto de geração baixo, uma vez que seu combustível é a água armazenada em seu reservatório, que em princípio é gratuita. Pode-se então utilizar esta energia no lugar da energia termoelétrica que possui alto custo de geração, devido principalmente aos preços dos combustíveis utilizados. Entretanto, a quantidade de água é limitada pela capacidade de armazenamento dos reservatórios e a produção de energia hidroelétrica depende dos seus níveis de armazenamento. Para gerenciar de forma sustentável esses recursos e ainda atender à demanda de energia são necessárias ferramentas para auxiliar o planejamento da operação de maneira a otimizar o uso dos recursos disponíveis.

O planejamento da operação energética de um sistema hidrotérmico consiste em estabelecer, para cada estágio do horizonte de planejamento, metas de geração para as usinas hidroelétricas e termoelétricas de forma a atender a demanda de energia e minimizar o valor esperado do custo de operação do sistema, ao longo do período de estudo, procurando-se também atender a determinados critérios de segurança no suprimento de energia. O custo de operação é representado no sistema pelos gastos com os combustíveis das usinas

termoelétricas e pelo custo de não atendimento à demanda de energia, ou seja, custo de déficit. O problema de planejamento da operação possui as seguintes características [19]:

- Estocástico: como o sistema de geração brasileiro é predominantemente hidráulico, a aleatoriedade das vazões tonrna o problema de planejamento da operação energética um problema estocástico dado às incertezas com relação às afluências futuras. O montante de recurso disponível nos reservatórios em estágios futuros é uma variável desconhecida, pois depende fortemente das afluências que ocorrerão nesses estágios.
- Acoplado no espaço: as usinas hidroelétricas são afetadas pela sua disposição ao longo de um rio, na maioria das vezes dispostas em cascata, onde a operação de uma usina a montante influencia a quantidade de recurso disponível para geração a jusante, já que a afluência desta depende do deplecionamento da anterior.
- Acoplado no tempo: a capacidade de armazenamento dos reservatórios das usinas hidroelétricas é limitada. Como a quantidade de energia gerada em um estágio influencia na quantidade disponível para os estágios futuros, existe uma relação de dependência entre a decisão de geração em um determinado período e as consequências futuras na operação.
- A necessidade de otimização multi-período e a grande quantidade de reservatórios torna o planejamento um problema de grande porte.

Nos sistemas hidrotérmicos devido à capacidade limitada de seus reservatórios somado às incertezas dos volumes de chuvas, e consequentemente das vazões afluentes futuras, observa-se uma relação entre a decisão tomada em um estágio qualquer e sua consequência futura. Em outras palavras, a decisão dos montantes despachados nas usinas hidroelétricas e termoelétricas é o fator fundamental na determinação do preço da energia e da garantia de seu fornecimento. Em um primeiro momento pode-se pensar que como a água armazenada nos reservatórios não possui um custo associado já que seu combustível é grátis, é melhor gerar toda a energia necessária através das usinas hidroelétricas, evitando os altos custos da geração termoelétrica. Se no presente for decidido minimizar o custo de geração esvaziando os reservatórios através de geração hidroelétrica para atender o mercado hoje e, futuramente ocorrer um baixo regime pluviométrico, provavelmente será necessário utilizar geração termoelétrica de alto custo para atendimento à demanda, aumentando o risco de corte de carga e interrupção do fornecimento de energia. Por outro lado, se for utilizado geração

termoelétrica para suprir a demanda hoje mantendo os níveis dos reservatórios elevados e ocorrerem altos índices pluviométricos no futuro, poderá haver vertimento no sistema, desperdiçando a oportunidade de gerar energia a preços baixos no futuro, e um aumento no custo da energia que poderia ter sido evitado.

Essas duas situações exemplificam as consequências do acoplamento temporal na operação do sistema hidrotérmico. A Figura 3-1 ilustra as possíveis decisões operativas e suas consequências de acordo com as afluências futuras. Na seção seguinte são apresentados os critérios para otimização do problema de planejamento da operação em sistemas hidrotérmicos.

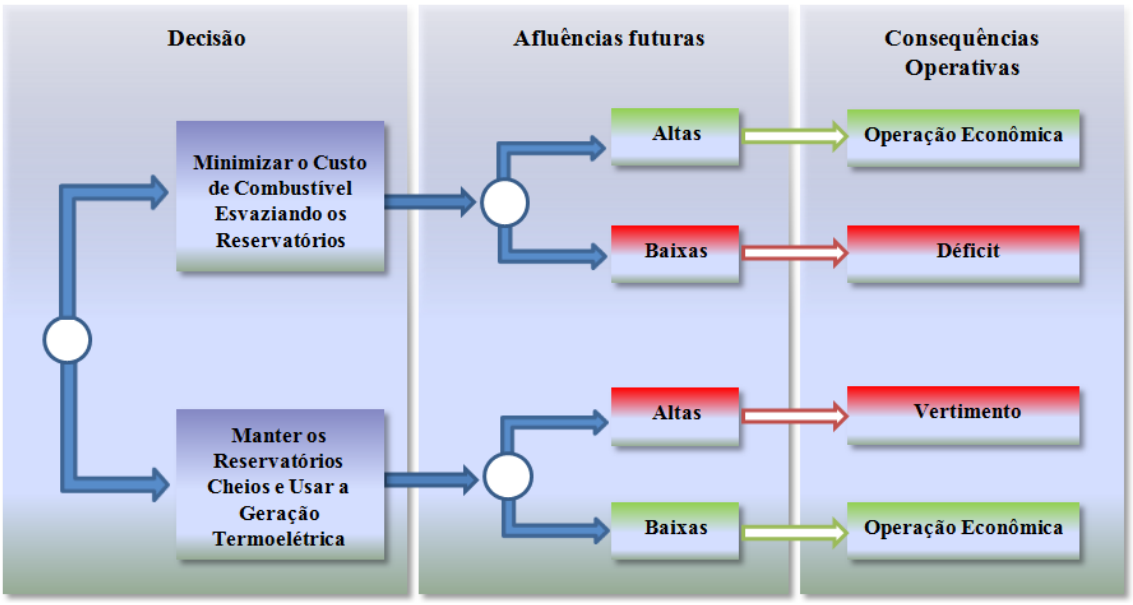


Figura 3-1. Esquema de decisões operativas em sistemas hidrotérmicos [19]

3.2 Otimização em Sistemas Hidrotérmicos

Em sistemas hidrotérmicos a água possui um valor associado que é medido em função do custo dos combustíveis das termoelétricas e dos custos de déficit que poderiam ser evitados no futuro. Como descrito anteriormente, o valor esperado do custo total de operação de sistemas hidrotérmicos (FCT) é composto pelo custo imediato de geração das usinas termoelétricas e do corte de carga somado ao valor esperado do custo futuro. Este último é composto pelo custo de geração das termoelétricas, pelo custo de déficit do sistema e pelo

custo da energia vertida que poderia ter sido gerada em estágios passados. O custo associado ao déficit representa as perdas que a interrupção do fornecimento de energia pode representar para o país, enquanto o custo da energia vertida representa o custo da energia termoelétrica que se teria economizado se a energia vertida tivesse sido gerada.

A Função de Custo Presente (FCP) representa o benefício do uso imediato da água, enquanto a Função de Custo Futuro (FCF) representa o benefício de armazenar a água no presente para uso em estágios futuros [19]. A Figura 3-2 apresenta estas duas funções, onde o eixo das abscissas representa o volume armazenado no final do estágio enquanto o eixo das ordenadas representa os valores das funções de custo presente e de custo futuro, expressos em unidades monetárias.

A FCP representa o custo de geração das usinas termoelétricas e corte de carga no estágio presente. Para volumes finais de armazenamento baixos, significa que a decisão durante o estágio foi esvaziar os reservatórios gerando pouca energia termoelétrica, e assim a FCP é baixa. A FCP aumenta à medida que o volume final de armazenamento aumenta, o que significa que para manter um alto nível dos reservatórios, foi necessário aumentar a geração de energia termoelétrica, aumentando assim a FCP.

A FCF por sua vez representa o custo da geração térmica e do custo de déficit do estágio atual até o final do período de estudo. A FCF tem o comportamento inverso à FCP, diminuindo com o aumento do volume armazenado no final do estágio. A decisão de economizar água no presente significa que no final do horizonte de planejamento os reservatórios estarão cheios, sendo necessário gerar menos energia termoelétrica, e consequentemente reduzindo o custo de operação. Além disso, a probabilidade de corte de carga diminui, minimizando um provável aumento na FCF devido ao custo de déficit. O custo aumenta à medida que os volumes nos reservatórios diminuem.

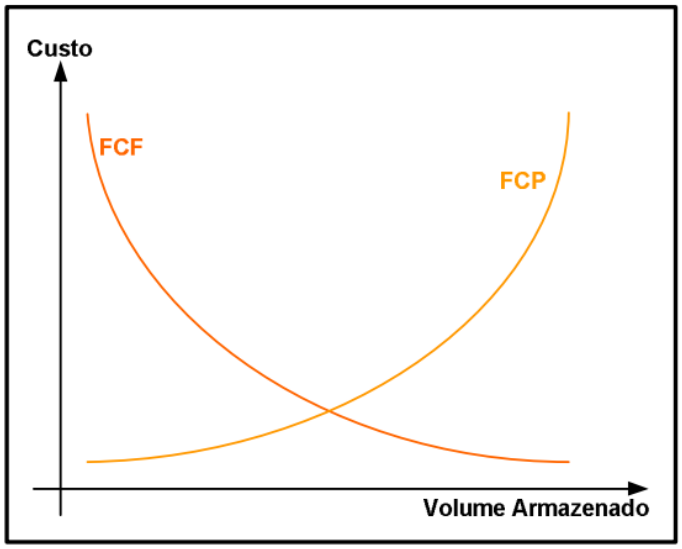


Figura 3-2. Funções de Custo Presente e Custo Futuro

O uso ótimo da água armazenada é representado pelo valor ótimo esperado para o custo total de operação de sistemas hidrotérmicos que é obtido no ponto que minimiza a soma do FCP e do FCF, ou seja $FCT_{mín}$ = min(FCP + FCF). O custo de operação mínimo é obtido no ponto em que a derivada da FCT é nula. É nesse ponto que as derivadas da FCP e da FCF são simétricas, iguais em módulo com sinais contrários. Estas derivadas são denominadas no problema como "valores da água".

$$\frac{\partial \mathrm{FCT}}{\partial \mathrm{V}} = \frac{\partial(\mathrm{FCP} + \mathrm{FCF})}{\partial \mathrm{V}} = \frac{\partial \mathrm{FCP}}{\partial \mathrm{V}} + \frac{\partial \mathrm{FCF}}{\partial \mathrm{V}} = 0 \rightarrow \frac{\partial \mathrm{FCP}}{\partial \mathrm{V}} = -\frac{\partial \mathrm{FCF}}{\partial \mathrm{V}} \quad (\mathbf{3.1})$$

A Figura 3-3 apresenta a FCT e o ponto de armazenamento que minimiza o custo global de operação.

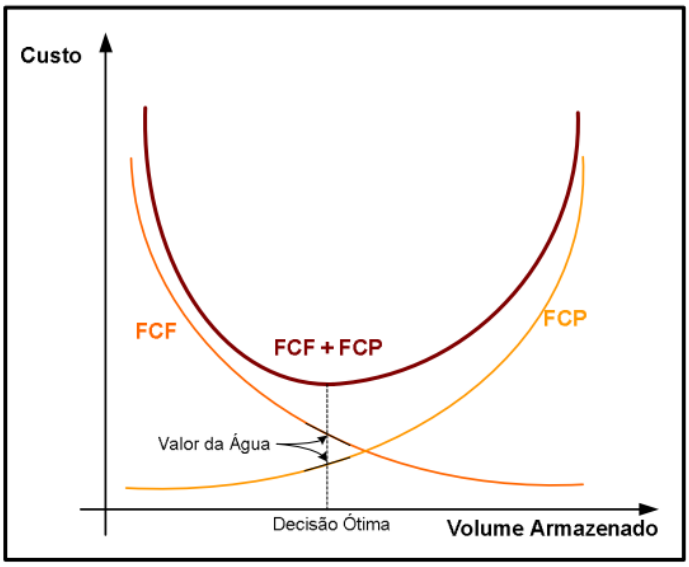


Figura 3-3: Uso ótimo da água

O valor da energia hidráulica (valor da água) é o valor da geração termoelétrica mais o déficit que se poderia substituir hoje ou no futuro. Este valor é resultante do processo de determinação da política ótima de operação. Com este conceito, pode-se considerar uma hidroelétrica como sendo uma termoelétrica cujo custo de operação é o valor da água, mas que varia de acordo com o nível de armazenamento dos reservatórios [17] [18] [19].

Depois do exposto nas seções anteriores, finalmente pode-se concluir que o planejamento da operação do SIN trata-se de um problema complexo e de grande porte devido à quantidade de reservatórios existentes, das restrições operativas das usinas termoelétricas e hidroelétricas, da incerteza nas afluências futuras da operação do sistema de forma interligada. Deve-se considerar a necessidade de estudar diferentes períodos de planejamento, com o objetivo de analisar os efeitos do planejamento a longo, médio e curto prazo. A seção seguinte descreve a cadeia de modelos energéticos utilizada no planejamento da operação do SIN dando ênfase em como o modelo de geração de séries sintéticas de vazões, objeto de estudo dessa dissertação, gera as vazões para os modelos de médio e curto prazo, NEWAVE, SUISHI e DECOMP.

3.3 Cadeia de Modelos do Planejamento da Operação do Sistema Brasileiro

Devido à complexidade envolvida no planejamento da operação energética do sistema de geração hidrotérmica do Brasil, são necessários modelos com diferentes horizontes de estudo e diferentes níveis de detalhamento dos elementos de geração e demanda considerados. O problema foi desagregado no tempo de acordo com as principais decisões a serem tomadas em cada etapa do planejamento, sendo dividido em subproblemas que podem ser resolvidos individualmente, onde os problemas de horizonte mais longo fornecem a função de custo futuro para os problemas de menor período (Figura 3-4).

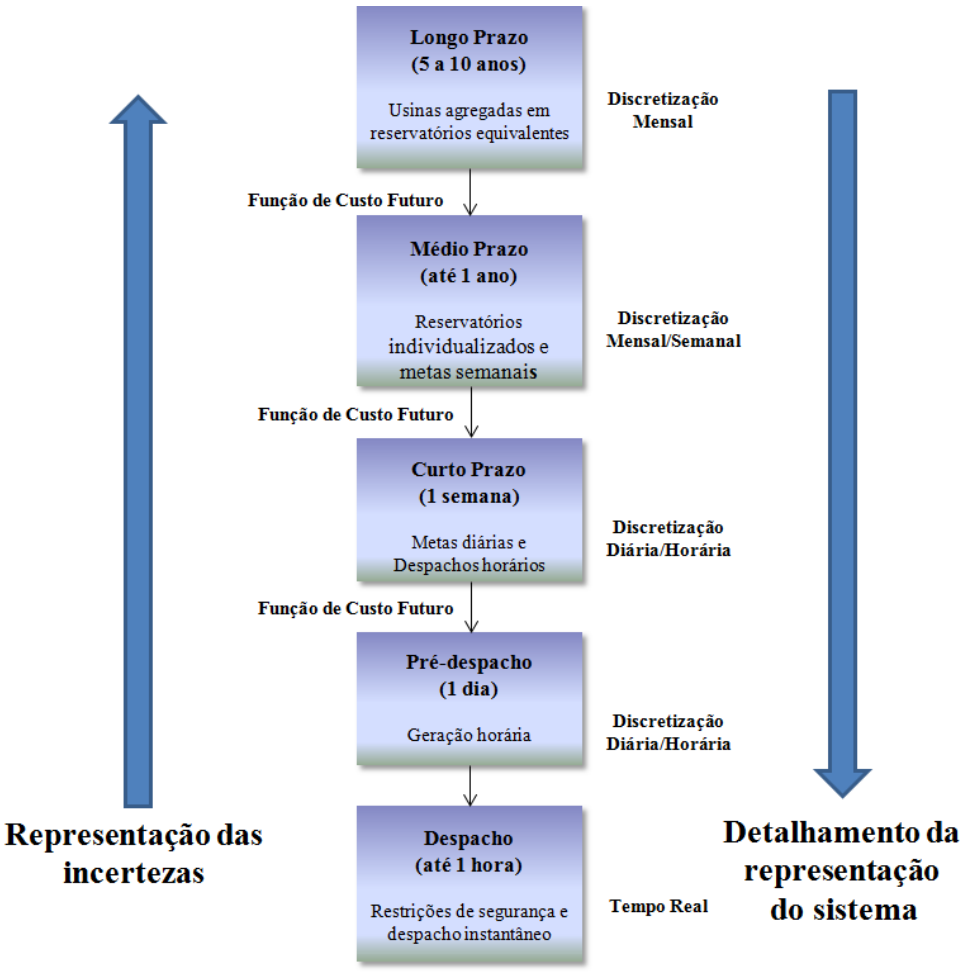


Figura 3-4: Horizontes do planejamento da operação

Uma breve descrição de cada etapa do planejamento é detalhada em [22] e descrita a seguir:

- *planejamento da operação de longo prazo:* nesta fase o horizonte de estudo é de cinco a dez anos discretizado em etapas mensais. Faz-se uma representação detalhada do processo estocástico das vazões afluentes aos reservatórios e as usinas hidroelétricas que compõem cada sistema são representadas de forma agregada (reservatórios equivalentes de energia - REE). Além disto, os sistemas podem trocar energia entre si até um limite máximo de intercâmbio entre macro-regiões. Desta etapa resulta uma função multivariada que define o valor econômico da energia armazenada em função dos níveis de armazenamento e afluência dos meses passados (tendência hidrológica), para cada estágio do período de planejamento, a **função de custo futuro;**
- *planejamento da operação de médio prazo:* o horizonte, neste caso, é de alguns meses e a incerteza relacionada às afluências aos reservatórios é representada através de uma árvore de vazões. Nesta etapa, as usinas são representadas de forma individualizada. O objetivo é, a partir da função de custo futuro gerada pelo modelo de médio prazo em um estágio que coincide com o final do horizonte do modelo de curto prazo, gerar uma função que retrate o valor econômico da água armazenada nos reservatórios em função dos níveis de armazenamento dos reservatórios;
- *programação diária da operação*: nesta etapa, o horizonte é de apenas alguns dias, discretizados em etapas horárias ou de meia em meia hora. Não é representada a incerteza das vazões. Em contrapartida, o parque hidrotérmico é representado de forma detalhada, levando-se em conta as restrições relativas as máquinas e turbinas, tais como: tomada e alívio de carga, faixas operativas das turbinas, entre outras. A rede de transmissão é representada com detalhes. A função de custo futuro gerada pelo modelo de curto prazo no estágio que coincide com último estágio do modelo de programação diária é utilizada para definir os valores da água, a partir dos quais será calculada a meta de geração de cada unidade geradora.

Tanto no planejamento a longo-prazo quanto no médio-prazo os modelos devem ser alimentados por dados associados à geração de cenários sintéticos de energias naturais afluentes para os sistemas equivalentes ou de vazões afluentes para cada usina hidroelétrica, de maneira que o comportamento estocástico das vazões seja bem representado nos modelos. O modelo que é utilizado atualmente para geração dos cenários sintéticos é o modelo

autorregressivo periódico de ordem p – PAR(p), que é capaz de representar de forma adequada as probabilidades de ocorrência de cada um dos cenários de afluência, considerando amostragem seletiva, e implementado no modelo GEVAZP desenvolvido pelo CEPEL [6] [23]. O GEVAZP faz parte de uma cadeia de modelos matemáticos computacionais para o planejamento da expansão da geração e da operação energética. A Figura 3-5 apresenta a cadeia dos modelos que são utilizados no planejamento e programação da operação [24] e desenvolvidos pelo CEPEL.

O modelo GEVAZP, foco dessa dissertação, é utilizado no horizonte de longo-prazo gerando cenários sintéticos de energia natural afluente (ENA) para os reservatórios equivalentes para o modelo NEWAVE, e de vazões afluentes para usinas individualizadas no modelo SUISHI. No curto-prazo, o GEVAZP fornece as vazões afluentes das usinas para o modelo DECOMP. Na seção seguinte é feita uma breve descrição desses modelos NEWAVE, SUISHI e DECOMP e como é feita pelo modelo GEVAZP a geração dessas séries sintéticas para esses modelos. Uma descrição de cada um dos modelos da cadeia é encontrada em [24].

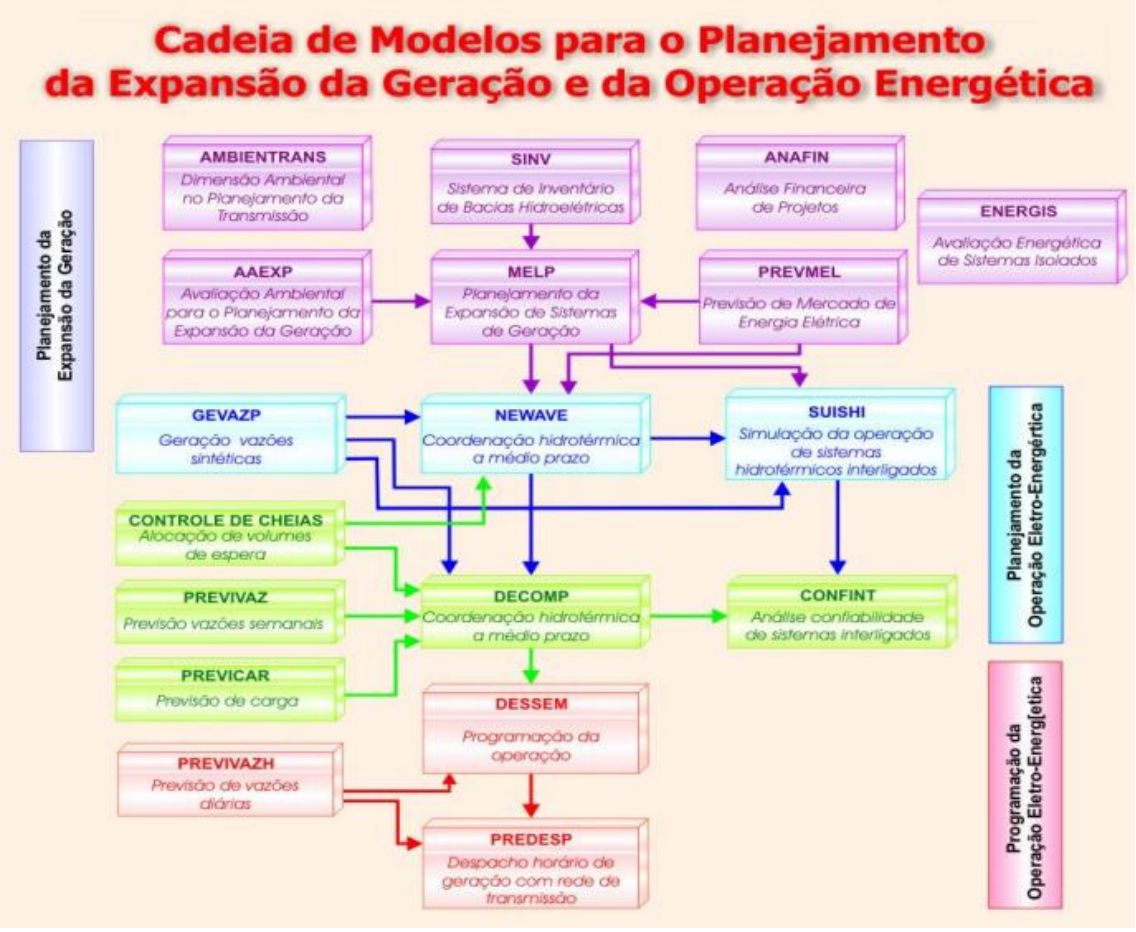


Figura 3-5: Cadeia de modelos utilizados no planejamento da expansão da geração e da operação energética [24]

3.4 Geração de séries sintéticas para os modelos NEWAVE, SUISHI e DECOMP

O modelo NEWAVE foi desenvolvido para o planejamento da operação de longo prazo, onde o período de planejamento pode variar de 5 a 10 anos, com discretização temporal de 1 mês. O objetivo principal do modelo NEWAVE é construir uma política de operação, com a qual se pode estabelecer uma alocação ótima dos recursos hídricos e térmicos de forma a minimizar uma combinação entre o valor esperado do custo de operação ao longo de todo o período de planejamento e um critério de segurança no suprimento, denominado *Conditional Value at Risk - CVAR* [25] [26]. Para tal, utiliza-se uma metodologia de agregação das usinas hidroelétricas em reservatórios equivalentes de energia [19] [27] [28].

O modelo SUISHI [29] é um modelo de simulação a usinas individualizadas da operação energética de sistemas hidrotérmicos interligados podendo ser acoplado ao modelo de decisão estratégica NEWAVE pela função de custo futuro para cada estágio e pode considerar restrições de operação locais decorrentes do uso múltiplo da água.

Tanto o modelo NEWAVE quanto o modelo SUISHI simulam um grande número de séries hidrológicas, calculando índices probabilísticos de desempenho do sistema para cada estágio da simulação. Estes diversos cenários de afluências estão em uma estrutura paralela (pente), (Figura 3-6.a) e são gerados pelo modelo GEVAZP.

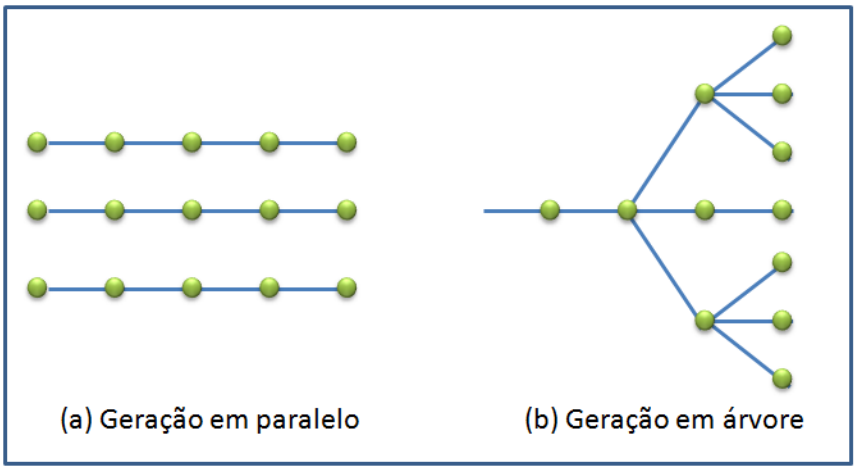


Figura 3-6: Geração de Séries - (a) em Paralelo, (b) em Árvore

O modelo DECOMP [30] [31] foi desenvolvido para aplicação no horizonte de curto prazo. Seu objetivo é determinar as metas de geração de cada usina de um sistema hidrotérmico, sujeito a afluências estocásticas, de forma a minimizar o valor esperado do

custo de operação ao longo do período de planejamento. Os cenários hidrológicos podem ser representados através de uma árvore de afluências (Figura 3-6.b), com probabilidades de ocorrência associadas a cada ramo.

Estes cenários são gerados a partir do modelo GEVAZP [6] [23] [32]. Na prática, o único cenário disponível são as afluências observadas do passado, chamadas de série histórica de vazões. Em geral, esse registro é insuficiente para representar uma amostra de tamanho necessário para estimar índices de risco aceitáveis. Entretanto as características básicas da série histórica podem ser capturadas por modelos estocásticos capazes de produzir séries sintéticas de afluências com probabilidades definidas no processo de geração das séries [23]. Entre os modelos probabilísticos em séries hidrológicas se destaca o modelo autorregressivo periódico, PAR(p) que ajusta para cada período da série um modelo autorregressivo. O modelo GEVAZP implementa o modelo PAR(p), descrito na próxima seção.

3.5 Modelo Autorregressivo Periódico

Séries hidrológicas de intervalo de tempo menor que um ano, tais como séries mensais, têm como característica o comportamento periódico das suas propriedades probabilísticas, como por exemplo, a média, a variância, a assimetria e a estrutura de auto-correlação. A análise deste tipo de séries pode ser feita pelo uso de formulações auto-regressivas cujos parâmetros apresentam um comportamento periódico. A esta classe de modelos costuma-se denominar modelos autorregressivos periódicos [33] [37], referenciados por modelos PAR(p), onde p é a ordem do modelo, ou seja, o número de termos autorregressivos do modelo. Em geral, p é um vetor, $p = (p_1, p_2, ..., p_{12})$, onde cada elemento fornece a ordem de cada período.

O modelo PAR($p_1, p_2, ..., p_{12}$) pode ser descrito matematicamente por:

$$\left(\frac{Z_t - \mu_m}{\sigma_m}\right) = \phi_1^m \left(\frac{Z_{t-1} - \mu_{m-1}}{\sigma_{m-1}}\right) + \ldots + \phi_{p_m}^m \left(\frac{Z_{t-p_m} - \mu_{m-p_m}}{\sigma_{m-p_m}}\right) + a_t \qquad (\mathbf{3.2})$$

onde:

Zt é uma série sazonal de período s

s é o número de períodos (s = 12 para séries mensais)

t é o índice do tempo, t = 1, 2, ..., N, função do ano T (T = 1, 2, ..., N) e do período m (m = 1, 2, ..., s)

N é o número de anos

$\boldsymbol{\mu_m}$ é a média sazonal de período s

$\boldsymbol{\sigma_m}$ é desvio-padrão sazonal de período s

$\boldsymbol{\varphi_i^m}$ é o i-ésimo coeficiente autorregressivo do período m

$\mathbf{p_m}$ é a ordem do operador autorregressivo do período m

$\mathbf{a_t}$ série de ruídos independentes com média zero e variância $\sigma_a^{2(m)}$

No DECOMP é interessante trabalhar com o menor número de cenários hidrológicos possíveis devido às restrições de tempo computacional, porém esta limitação pode não ser suficiente para caracterizar bem o processo estocástico. Para solucionar esse problema, empregou-se no modelo GEVAZP a análise de agrupamentos aplicadas à um grande número de cenários hidrológicos gerados. A técnica de agrupamento proporciona a escolha de um conjunto representativo de cenários, onde o conjunto representativo de cenários hidrológicos irá conter toda a informação necessária para representar, dentro de uma dada precisão, o processo estocástico de vazões, preservando as correlações temporais e espaciais.

3.6 Agrupamento dos cenários hidrológicos

O principal objetivo da aplicação das técnicas de agregação do modelo GEVAZP é a redução do número de cenários hidrológicos através da escolha de um conjunto cenários representativos do conjunto original de cenários hidrológicos. Os cenários são obtidos através

de agrupamento de cenários semelhantes que possuem características similares aos demais componentes do grupo em que estão localizados e têm pesos diferentes, de acordo com o número de cenários contidos no grupo. A cada estágio do horizonte de estudo e a cada ramo da árvore de afluências são gerados M cenários hidrológicos distintos, por exemplo, M=1000, e equiprováveis para o processo de agregação. O número de aberturas em cada estágio é o número de K agrupamentos obtidos a partir dos M cenários, onde K é um dado de entrada. O método escolhido foi o método de agrupamento não hierárquico K-means [34]. O algoritmo K-means pode ser dividido em 3 partes:

- Inicialize os K grupos através do sorteio de K centróides iniciais;
- Calcule a distância Euclidiana [35] entre cada objeto M e os K grupos e realoque o objeto no grupo com menor distância;
- Repita o passo anterior até o momento em que nenhum objeto seja mais realocado.

No GEVAZP o processo de agregação das séries de afluência é inicializado através do sorteio de sementes aleatórias para representar os centróides dos K grupos [23], com distribuição *log-normal* para evitar a geração de séries negativas [6]. Para cada iteração do processo de convergência, é calculada a distância de Mahalanobis [36] (no lugar da Euclidiana) entre o objeto (vazão) e o centróide de cada grupo, e o objeto é realocado no grupo com menor distância (mais semelhante) e os centróides dos grupos são então recalculados. Após a convergência do processo, os cenários representantes serão os objetos mais próximos ao centróide do grupo, onde cada cenário representante tem associado a ele uma probabilidade de ocorrência, que é calculado através do número de objetos alocados no grupo dividido por M.

A substituição da distância Euclidiana pela de Mahalanobis no processo de agregação imprime um aumento de complexidade no cálculo da distância e consequente aumento de tempo computacional. A distância Euclidiana é a distância entre dois pontos no espaço Euclidiano n-dimensional, que pode ser provada pela aplicação repetida do teorema de Pitágoras. A distância Euclidiana entre os pontos $P = (p_1, p_2, \dots p_n)$ e $Q = (q_1, q_2, \dots q_n)$, num espaço n-dimensional é definida como [35]:

$$D_E\ (P,Q) = \sqrt{(p_1 - q_1)^2 + \cdots + (p_n - q_n)^2} = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2} \qquad (\mathbf{3.3})$$

Já a distância de Mahalanobis leva em consideração a correlação entre os elementos de cada vetor. Essa abordagem é a mais apropriada ao modelo GEVAZP pois preserva a correlação espacial entre as vazões de cada posto.

Formalmente, a distância de Mahalanobis entre um grupo de valores com a média $\mu = (\mu_1, \mu_2, \ldots, \mu_n)$ e a matriz de covariância $\boldsymbol{S}$ para um vetor n-dimensional $x = (x_1, x_2, \ldots, x_n)$ é definida como [36]:

$$D_M\ (x,\mu) = \sqrt{(x - \mu)^T S^{-1} (x - \mu)} \qquad (\mathbf{3.4})$$

A distância de Mahalanobis é equivalente à Euclidiana (Equação 2.1) quando a correlação entre os pontos é nula, pois a inversa da matriz de covariância ($\boldsymbol{S^{-1}}$) é a própria matriz identidade.

Em termos computacionais, em um espaço N-dimensional, o número de operações de ponto flutuante necessárias para calcular o quadrado da distância Euclidiana é **3N - 1** (1 subtração, 1 multiplicação e uma soma para cada elemento do vetor, exceto o último) ou seja, o cálculo tem complexidade **O(N)**. Já o número de operações para calcular o quadrado da distância de Mahalanobis é dado por **N + ($2N^2$ – N) + 3N - 1** (subtração, multiplicação vetor-matriz e produto interno vetor-vetor), ou seja, **$2N^2$ + 3N –** 1, e portanto complexidade O**(N^2)**. Este aumento de complexidade no cálculo das distâncias N-dimensionais, se reflete na forma de uma função quadrática no aumento do tempo computacional do processo de agregação. Em um vetor de dimensão N = 100 por exemplo, o tempo cresce de 299 unidades de tempo para 20.299. Assim, os cálculos das distâncias entre as vazões e centróides dos grupos no processo de agregação de cenários são avaliados na seção seguinte como potenciais procedimentos para uma possível estratégia de paralelização do modelo GEVAZP. Este procedimento envolve cálculos envolvendo álgebra de vetores e matrizes, adequados à paralelização em GPUs.

Além disso, também é avaliada a paralelização da geração das séries sintéticas, uma vez que as aberturas são independentes dentro de um mesmo estágio.

3.7 Estratégias de Paralelização do modelo GEVAZP

No processo de paralelização de um *software* deve-se considerar alguns aspectos:

- conhecimento do problema teórico
- particionamento do problema em funcionalidades e/ou em domínio
- granularidade do problema
- conhecimento do código fonte do programa
- tipo de paralelização levando em consideração o tempo de comunicação entre os processos (troca de mensagens MPI ou memória compartilhada OpenMP, CUDA, OpenCL, etc.)
- acesso à disco (I/O), escalabilidade entre outros.

Nas próximas seções são abordados alguns desses pontos relativos ao modelo.

3.7.1 Estudo da metodologia do problema

Após o estudo da metodologia utilizada pelo GEVAZP para geração de vazões sintéticas observou-se a característica de que as vazões são dependentes no tempo, ou seja, uma vazão Z_t gerada para o período *t* utiliza as vazões passadas geradas nos períodos *t - p, t – p - 1... t -1*, onde p é a ordem do modelo PAR(p) definido. Entretanto observa-se a independência entre as aberturas de um mesmo período tanto na geração em paralelo quanto na geração em árvore, como pode ser observado na Figura 3-7.

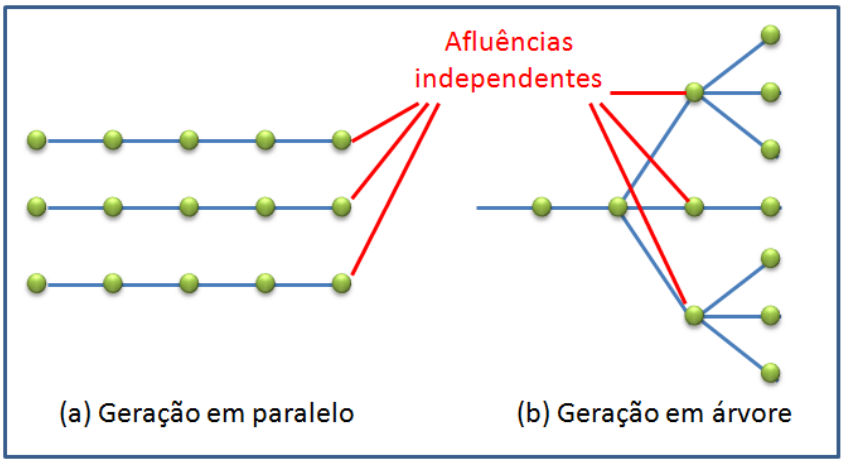


Figura 3-7: Afluências independentes no tempo

Essa característica pode ser explorada como uma das estratégias de paralelização do modelo GEVAZP. Para cada período *t*, uma vez que as tarefas realizadas na geração de um cenário de afluência de cada uma das aberturas são independentes entre si, pode-se aplicar técnicas de processamento paralelo resolvendo-se o problema em diversos processadores simultaneamente.

Outra característica relevante do modelo na decisão da estratégia de paralelização está na técnica de agregação de vazões utilizado. Como descrito nas seções 3.6 e 3.7, para cada ramo da árvore são gerados M cenários hidrológicos equiprováveis e agrupados em K grupos, onde K é o número de aberturas de cada estágio (Figura 3-6 e Figura 3-7).

O método de agrupamento K-means empregado utiliza a similaridade entre os cenários hidrológicos para agrupá-los em K grupos, calculando a distância de Mahalanobis (no lugar da Euclidiana) entre os cenários e os centróides dos grupos. Após os cálculos das distâncias, os objetos (cenários de vazões) são reagrupados de acordo com o grupo mais semelhante e os centróides são recalculados até o processo convergir. Esse processo possui alguns pontos independentes como o cálculo das distâncias, cálculo das matrizes de covariância e cálculo dos centróides, onde podem ser aplicadas as técnicas de processamento paralelo, resolvendo-se parte desses cálculos em várias processadores simultaneamente.

3.7.2 Estudo do Algoritmo de Agregação

O algoritmo de agregação utilizado no modelo GEVAZP é uma variação do algoritmo publicado em 1979 por Hartigan & Wong [34]. Nesta publicação os autores apresentam uma versão mais eficiente para o já conhecido algoritmo K-means. Como descrito na seção 3.6, a idéia principal do algoritmo é classificar M pontos de dimensão N, em um conjunto de K grupos de maneira que a soma dos quadrados dos pontos dentro de cada grupo seja minimizada. Na prática o que se faz é calcular a distância entre cada ponto ***m*** pertencente a M e o valor médio de cada grupo ***k*** (centróide), e mover o ponto parar o grupo com menor distância. A cada ponto movimentado, é recalculada a média dos dois grupos envolvidos na movimentação. O algoritmo converge quando nenhum movimento de um ponto para outro grupo é capaz de reduzir a soma dos quadrados dentro dos grupos.

A versão de Hartigan & Wong tem como idéia chave a criação de um conjunto de grupos ativos, atualizado a cada iteração. Um grupo está ativo quando alguma movimentação de seus pontos é realizada. Não é necessário recalcular a distância para os grupos que não estão ativos, pois como não houve nenhuma movimentação, seu centróide permanece o mesmo. A cada iteração calculam-se as distâncias dos M pontos apenas para os grupos pertencentes ao conjunto ativo. Esta abordagem minimiza a quantidade de processamento do algoritmo.

Esse procedimento pode ser dividido em três partes principais:

- **Rotina de Inicialização** – são gerados centróides aleatórios para cada grupo, e então os dados das M observações são alocados nos grupos, de acordo com as distâncias calculadas entre os M objetos os K centróides gerados;
- **Rotina OPTRA** (*Optimal transfer stage)* – calculam-se as distâncias dos M pontos para os K grupos pertencentes ao conjunto ativo, atualizam-se os centróides e o conjunto de grupos ativos. Essa rotina armazena os dois centróides mais próximos de cada observação *I* pertencente a M.
- **Rotina QTRAN** (*Quick transfer stage)* – após as atualizações dos centróides na rotina OPTRA, recalcula-se para cada ponto, a distância para os dois grupos armazenados na rotina OPTRA, e faz as movimentações e atualizações necessárias nos grupos.

O algoritmo publicado em 1979 [34] é descrito em detalhes no ANEXO A.

3.7.3 Modificações no Algoritmo de Hartigan & Wong

Na versão serial do modelo GEVAZP, foi necessária uma mudança no algoritmo original de Hartigan & Wong. A métrica de similaridade no método original é dada pela distância Euclidiana entre cada ponto ***i*** e cada grupo ***k***. No modelo GEVAZP, a métrica utilizada para avaliar a similaridade entre as vazões e os centróides dos grupos é a distância de Mahalanobis.

Outra mudança no algoritmo original é no que diz respeito à atualização das médias dos grupos após ocorrer alguma movimentação de séries entre eles. No algoritmo descrito na

seção 0, a atualização ocorre nos passos 4a, 4b e passo 6. A cada passo no laço de M objetos, a atualização causa uma dependência entre o passo i e o passo i + 1. Ou seja, dado que um objeto *I = M [i]* foi transferido de um grupo *L1* para um grupo *L2* no passo i, o valor médio dois grupos são atualizados neste passo. No passo i + 1, as duas novas médias de L1 e L2 serão consideradas para o cálculo das distâncias do objeto *I = M [i + 1]* e os grupos. Este comportamento imprime uma dependência serial, não se permitindo assim paralelizar o algoritmo de agregação.

Com o intuito de viabilizar a paralelização do algoritmo, foi proposto que a atualização dos dois grupos envolvidos em uma transferência a cada iteração, fosse substituída por uma atualização das médias de todos os grupos, fora do laço de M objetos. Dessa forma os valores dos centróides permanecem o mesmo durante os M passos do laço, e o cálculo das distâncias pode ser distribuído para os diferentes processadores. Ao final da iteração, todos os centróides são atualizados de acordo com os objetos atribuídos a cada grupo.

Para avaliar o impacto dessa mudança no algoritmo, foi realizado um estudo com esta nova versão para avaliar se a "qualidade" das séries geradas permaneceu em um nível aceitável ou se houve degradação a ponto de invalidar tal abordagem.

3.7.4 Testes de validação do algoritmo de agrupamento modificado

O modelo GEVAZP possui um módulo de testes para avaliar se as séries geradas estão de acordo com certos parâmetros estabelecidos em sua metodologia. Em [23] e [32] a autora descreve os ajustes necessários no modelo PAR(p) proposta por [37]. Nesta metodologia a estratégia de seleção do modelo é dividida em três partes: identificação do modelo, estimação do modelo e verificação do modelo. A terceira etapa é responsável por realizar testes estatísticos com o objetivo de verificar se as hipóteses consideradas durante as 2 etapas anteriores são atendidas. Estes testes são melhores detalhados em [23] e [32].

O conjunto de dados escolhido para validação foram os utilizados no Planejamento Mensal da Operação (PMO). Trata-se de 12 casos de Janeiro a Dezembro, onde para cada mês é definido um número diferente de agrupamentos. Os 12 PMOs foram executados com a versão serial original (Versão 1) e com a versão serial modificada, onde os centróides são

atualizados após o laço das M séries (Versão 2). Os casos executados com a versão modificada passaram em todos os testes realizados pelo modelo.

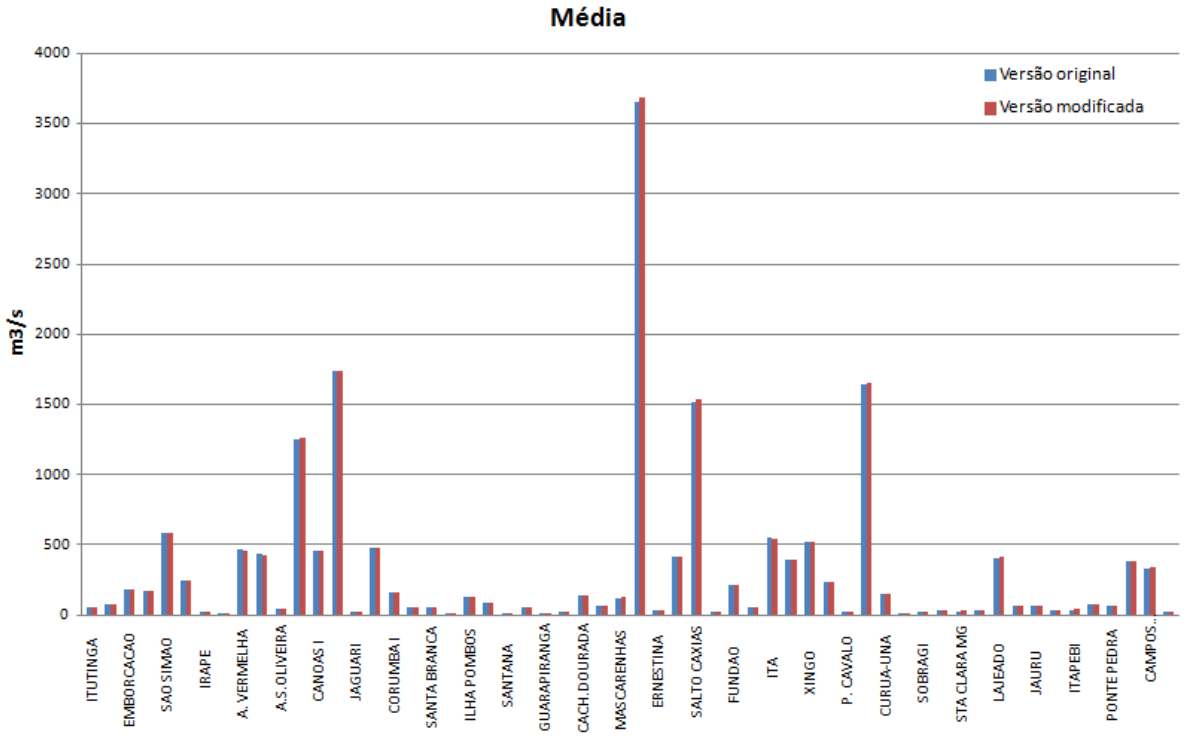


Figura 3-8: Comparação entre as médias das vazões da versão original e da modificada

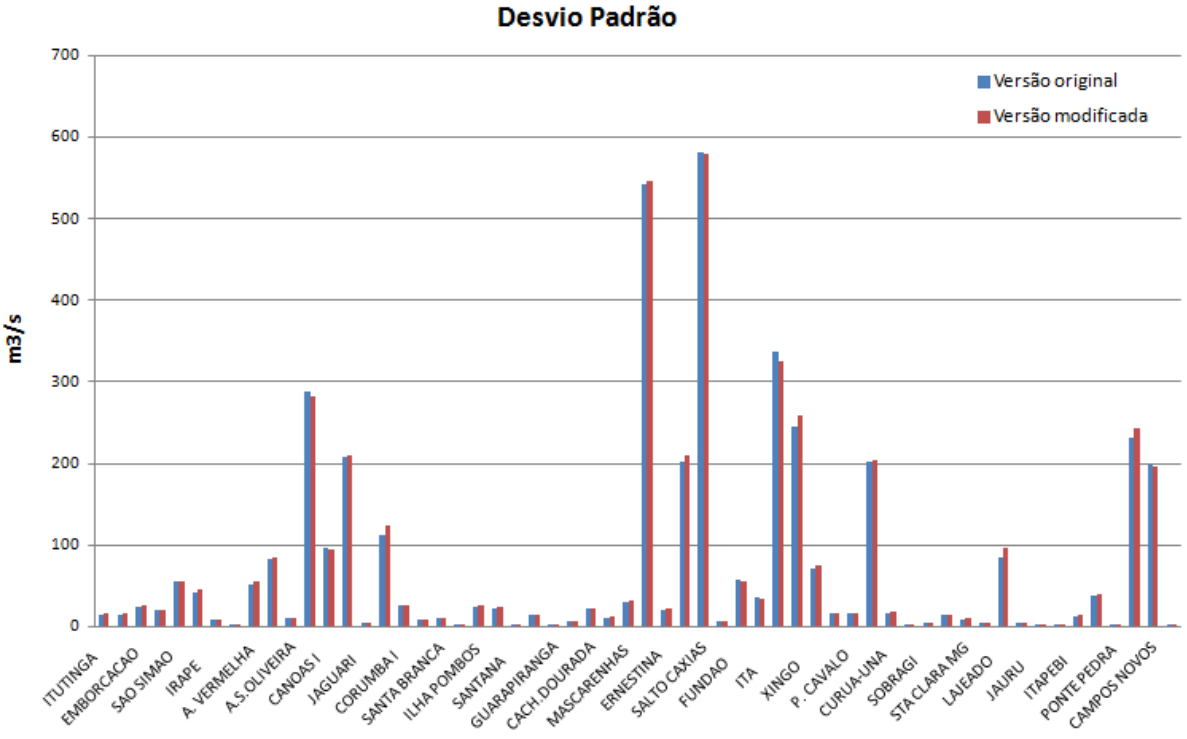


Figura 3-9: Comparação entre os desvios padrão das vazões versão original e da modificada

A Figura 3-8 e a Figura 3-9 apresentam uma comparação entre as médias e os desvios padrão das vazões geradas pela versão original e pela modificada. Pode-se perceber o valor das médias e dos desvios da nova versão obteve valores bem próximos da versão original.

Outro ponto a ser validado é a variação do número de iterações e o tempo necessário para a convergência do algoritmo na Versão 2. A Tabela 3-2 apresenta o número de iterações e o tempo de convergência da Versão 1 e da Versão 2, para os 12 meses estudados. Pode-se observar que na versão modificada (Versão 2) alguns casos demoram mais para convergir, ou seja, possuem mais iterações. Especificamente nos meses de Agosto e Setembro o número de iterações praticamente triplicou juntamente com o tempo de execução. Esse é um comportamento causado pela atualização dos centróides apenas no final do laço de M objetos. Verifica-se uma tendência de aumento maior nos meses que possuem mais grupos. Este comportamento não invalida o uso da Versão 2 mas insere uma necessidade de que o desempenho da implementação paralela seja suficiente bom para compensar esse tempo adicional.

Deve ser objeto de pesquisa futura encontrar o ponto ideal no algoritmo em que a atualização dos centróides dos grupos deve ser feita de forma a minimizar o número de iterações, e consequentemente o tempo de execução, e ainda ser possível manter o algoritmo paralelizável. Na versão original, este ponto é a atualização a cada iteração, ou seja, um limite inferior. Na versão modificada, no final do laço dos M objetos, ou seja, um limite superior. Deve-se encontrar então um ponto intermediário ótimo que minimize o tempo computacional sem degradar o desempenho da versão paralela. Este estudo foge ao escopo deste trabalho, ficando como sugestão para trabalhos futuros.

Baseados nesses resultados conclui-se a degradação verificada das séries geradas com o novo algoritmo foi insignificante, sendo estas equivalentes em termos probabilísticos à Versão 1, validando o seu uso.

Tabela 3-2: Número de iterações e tempo total de execução da versão serial original (Versão 1) e da versão serial modificada (Versão 2)

		Versao 1 (Original)		Versao 2	
Meses	**No Clusters**	**Tempo (s)**	**Iteracoes**	**Tempo(s)**	**Iteracoes**
jan	136	27.87	9	31.45	8
fev	116	20.45	6	27.67	10
mar	143	22.69	6	34.23	9
abr	143	22.71	6	34.33	9
mai	193	31.41	9	54.97	15
jun	267	34.93	7	70.04	14
jul	513	47.07	6	186.62	21
ago	353	41.16	7	119.03	21
set	303	36.66	6	88.16	18
out	251	33.87	6	74.58	15
nov	228	31.60	6	61.19	13
dez	152	30.46	10	35.55	9

3.8 Particionamento do problema, granularidade e modelo de paralelização para cada parte do problema

Ao realizar testes na execução da versão serial do modelo GEVAZP variando o número de aberturas em um determinado período, observaram-se as seguintes características:

- Casos com poucas aberturas apresentam baixo tempo computacional;
- Casos com muitas aberturas podem apresentar tempos computacionais elevados de acordo com o método de agregação de cenários: para um caso com cerca de 1000 aberturas o tempo computacional da estrutura de cenários paralela (Figura 3-6.a) foi de apenas alguns poucos minutos enquanto que na estrutura em árvore com agregação em grupo (Figura 3-6.b) o tempo medido foi de mais de uma hora;
- Os tempos medidos para geração de cada vazão em um período é da ordem de segundos;
- Os tempos medidos para cada iteração do agrupamento K-means é da ordem de centésimos de segundo.

A partir dessas métricas derivam-se as seguintes conclusões:

- A granularidade do processo de agrupamento é pequena, uma vez que em uma versão paralela do modelo, a atualização dos dados dos processadores deve ser feita na ordem de centésimos de segundo. Para casos de granularidade baixa, o tipo de paralelização mais apropriado é a divisão de tarefas em processos com memória compartilhada através de **CUDA** uma vez que o tempo para troca de mensagens MPI entre os processadores poderia ser maior que o próprio tempo de processamento entre cada iteração.
- A granularidade do processo de geração de vazões por sua vez é maior, uma vez que de acordo com o número de aberturas, o tempo de atualização dos dados entre os processadores varia na ordem de segundos para os casos com poucas aberturas, até centenas de segundos no caso de muitas aberturas (acima de 1000). Neste caso de granularidades mais altas, é possível utilizar a troca de mensagens via **MPI** com um menor comprometimento do desempenho, podendo-se utilizar também as características de memória compartilhada em computadores com mais de um núcleo de processamento a fim de explorar as vantagens desse tipo de paralelização.

De posse dessas observações, decidiu-se pela a divisão deste trabalho em duas linhas de pesquisa: (i) paralelização das afluências em memória distribuída, utilizando MPI, e (ii) paralelização do método de agregação em memória compartilhada, utilizando a plataforma CUDA. Através das duas implementações será possível comparar os benefícios de cada uma das filosofias de programação paralela, as vantagens e limitações de cada uma delas, e quais se adequam mais a cada tipo de problema.

3.8.1 Estratégia 1 – Utilização da biblioteca MPI

Paralelizar a geração de afluências em árvore com agregação de cenários utilizando-se a troca de mensagens MPI, por dois motivos principais:

- No trabalho [38], diferentes algoritmos de agrupamento de cenários podem ser usados na geração das séries de afluências. Uma vez que o tipo de agregação utilizada no GEVAZP é apenas uma das possíveis técnicas utilizadas para escolher os cenários mais representativos, é importante paralelizar a geração de cenários do

modelo ficando assim a versão paralela independente do tipo de agregação, permitindo ainda a viabilidade de estudo de outras técnicas de agrupamento.

- A utilização da biblioteca de troca de mensagens MPI para avaliação do algoritmo paralelo em *cluster* de computadores com memória distribuída permite executar o programa com mais processadores que com técnicas de memória compartilhada (OpenMP), onde será possível avaliar a eficiência e a escalabilidade do algoritmo para um grupo maior de núcleos de processamento.

Assim na paralelização em MPI, optou-se pelo particionamento do domínio do problema no nível das aberturas das séries de afluência, onde os N cenários são gerados em paralelo pelos processadores de um cluster. Após a geração dos N cenários de um determinado período, as informações geradas em cada um dos processadores têm de ser consolidadas e enviadas a todos os processadores para serem utilizadas na geração do próximo período.

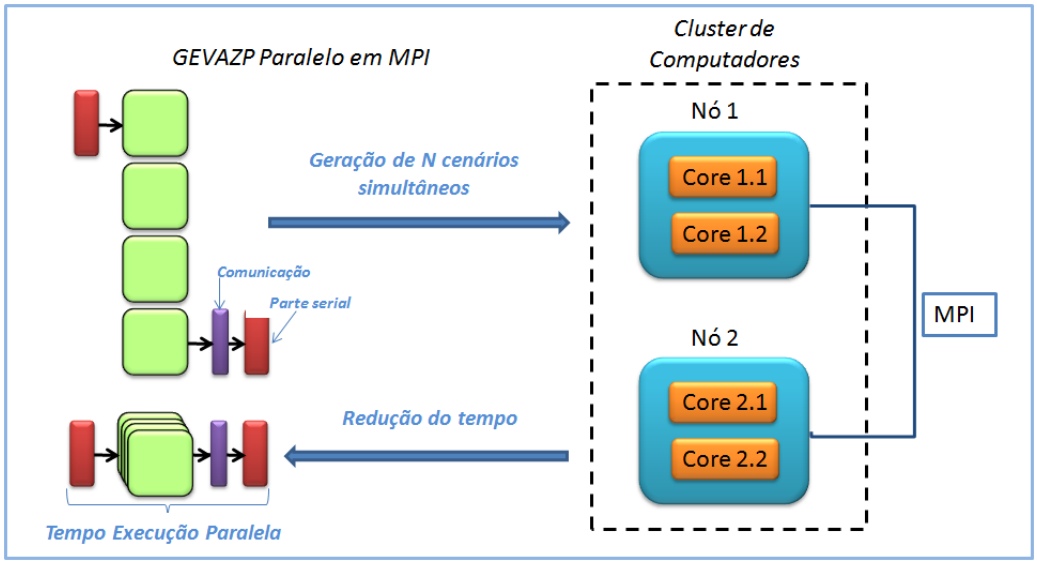


Figura 3-10. Paralelização em MPI

3.8.2 Estratégia 2 – Utilização da plataforma CUDA

Paralelizar o algoritmo de agrupamento das séries geradas, utilizando-se a plataforma CUDA. Apesar da geração de séries do modelo GEVAZP ser independente do algoritmo de

agrupamento, esta versão do algoritmo K-means [34] foi modificada para considerar a correlação entre as séries geradas, é atualmente utilizada oficialmente no setor elétrico na geração das séries para os modelos NEWAVE, SUISHI e DECOMP. Dessa forma existe uma forte tendência de que este algoritmo seja mantido nos próximos anos, o que motiva à pesquisa de paralelização do algoritmo de agrupamento. A escolha da plataforma CUDA foi feita pelos seguintes motivos:

- O algoritmo de agrupamento por ter uma baixa granularidade é mais adequado a técnicas de programação que utilizam o conceito de memória compartilhada;
- Crescente utilização da plataforma CUDA na busca de desempenho de aplicações científicas paralelas;
- Alto poder de processamento das placas NVIDIA, a um custo bem mais baixo em relação a um *cluster* de computadores;
- Oportunidade de investigar uma nova linha de pesquisa em processamento paralelo e integrar aplicações em CUDA com as aplicações em MPI.

Na paralelização utilizando a plataforma CUDA, optou-se pelo particionamento do domínio do problema no cálculo das distâncias entre as vazão e os centróides dos grupos, na agregação dos cenários de vazões. Diferente de outras abordagens paralelas do algoritmo K-means, o foco desta implementação é no cálculo da similaridade entre os objetos, devido à distância de Mahalanobis ter um custo computacional muito maior que a distância Euclidiana.

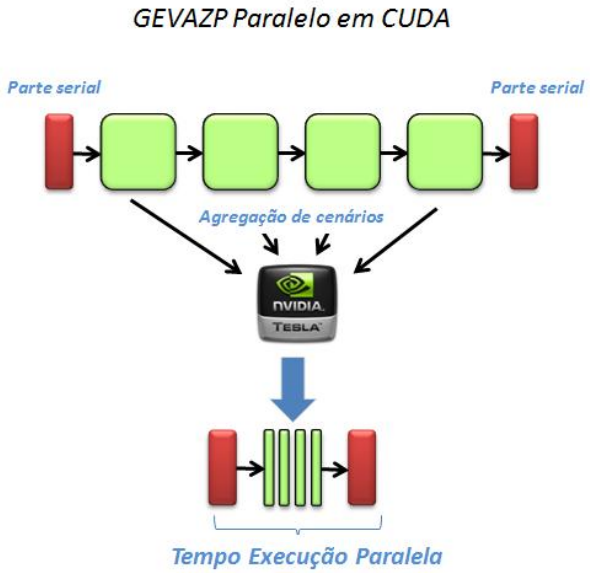


Figura 3-11: Paralelização em CUDA

A distância Euclidiana é calculada através de duas operações matriciais: diferença entre dois vetores *a* e *b*, resultando em um vetor *c*. Em seguida calcula-se o produto interno entre o vetor *c* e c^T.

A distância de Mahalanobis é calculada através de três operações matriciais:

- Diferença entre dois vetores *a* e *b*, resultando em um vetor *c*.
- Multiplicação do vetor *c* pela inversa da matriz de covariância $\boldsymbol{S^{-1}}$, resultando em um vetor *d*.
- E por fim calcula-se o produto interno entre o vetor *d* e c^T.

A multiplicação adicional pela matriz $\boldsymbol{S^{-1}}$ insere um custo computacional considerável, levando a tempos computacionais muito maiores quando comparada com a utilização da distância Euclidiana, principalmente com um número elevado de cenários M e *grupos* K. Assim, o foco desta implementação está na paralelização do cálculo das distâncias de Mahalanobis.

Na versão em CUDA, os cálculos das distâncias são distribuídos pelas diferentes *CUDA-threads* criadas na chamada da função e gerenciadas pela GPU. Para tal, foram avaliados diferentes algoritmos explorando paralelismo no nível de tarefa (*TLP*), variando-se também o número de *threads* para cada tipo. Estudaram-se os impactos no desempenho do algoritmo explorando os diferentes tipos de acesso às memórias da GPU, implementando versões utilizando memória global, memória compartilhada e registradores. Além disso foi feita também uma avaliação do uso da biblioteca CUBLAS, da NVIDIA .

4 GEVAZP PARALELO EM CUDA

Uma vez introduzida a tecnologia das placas GPU, o conceito de GP-GPU, o modelo de programação CUDA, e definidas as estratégias de paralelização do modelo GEVAZP, este capítulo descreve a implementação em CUDA do algoritmo de agrupamento das séries de vazões e energias geradas pelo modelo GEVAZP. O cálculo das distâncias entre cada vazão e os centróides dos grupos, utilizado no algoritmo de agregação, é ponto principal a ser explorado nessa metodologia paralela. A base desta implementação é realizar os cálculos das distâncias em paralelo na GPU, utilizando os resultados na parte serial, na CPU.

Após a modificação feita no algoritmo original (seções 3.7.3 à 3.7.4), cada distância pôde ser calculada independente das demais. Na distância de Mahalanobis, métrica utilizada no GEVAZP para medir a similaridade entre os cenários de vazões e os centróides dos grupos, são utilizadas operações vetoriais como subtração e produto interno entre vetores, e também a multiplicação entre vetores e matrizes. Tais operações possuem um grande potencial a ser explorado na arquitetura das GPUs. O processo de desenvolvimento da versão paralela em CUDA foi realizado através de versões incrementais, ou seja, otimizações feitas nas primeiras versões foram incorporadas nas versões seguintes. As duas primeiras versões exploram a utilização dos diferentes tipos de memória e uma terceira versão avalia o uso da biblioteca CUBLAS [7].

A versão inicial aloca as estruturas de dados somente na memória global do *device*. Esta versão apenas traduziu os cálculos que eram antes realizados na CPU para o código em GPU. Devido à latência de acesso à memória global ser muito grande, verificou-se a importância de minimizar o acesso a este tipo de memória. Desta forma, através do estudo do algoritmo, foi possível modificá-lo de forma a otimizar o acesso aos dados em busca de um melhor desempenho.

Em seguida, foi desenvolvida uma versão utilizando-se memória compartilhada, cuja latência de acesso é cerca de duas ordens de grandeza menor que a da memória global. O algoritmo foi completamente modificado visando maximizar a colaboração entre as *CUDA-threads* e um melhor uso dos recursos da GPU. Por último, foi desenvolvida uma versão utilizando-se a biblioteca paralela de álgebra linear CUBLAS para realizar parte dos cálculos entre vetores e matrizes, com o objetivo de avaliar o ganho de desempenho que esta biblioteca pode oferecer à aplicação.

As seções seguintes descrevem em detalhes as implementações em CUDA de cada uma destas versões, as partes comuns entre elas e a estratégia de implementação utilizada.

4.1 Implementação da solução paralela em CUDA

O código serial da rotina de agregação é codificado em FORTRAN. Como descrito no ANEXO A, ela pode ser dividida em 3 partes principais:

a) Inicialização dos K centróides;
b) Rotina OPTRA;
c) Rotina QTRAN.

As três rotinas calculam as distâncias entre as séries e os grupos. Na inicialização dos grupos, são calculadas K*M distâncias. Na rotina OPTRA, são calculadas as distâncias entre as M séries e os grupos pertencentes ao conjunto de "grupos ativos" (menor que K). E na rotina QTRAN são recalculadas para cada série as distâncias entre os dois grupos C1 e C2 onde elas estão alocadas.

A estratégia de implementação é calcular em paralelo na GPU as distâncias necessárias em cada uma das rotinas, e então usar as distâncias no código serial na CPU. Para tal, foi criada na linguagem C a rotina *cuda_mahala()* (Figura 4-1). Ela é o ponto de comunicação entre a CPU (*host*) e a GPU (*device*). O código FORTRAN realiza a chamada à *cuda_mahala()* passando como parâmetro as estruturas de dados necessárias. A função *cuda_mahala()* utilizada as seguintes estruturas de dados:

- nSeries – inteiro com o número de séries geradas – M séries;
- nClusters – inteiro com o número de grupos – K grupos;
- nsis – inteiro com a dimensão dos vetores das séries e dos grupos – dimensão N.
- stip – vetor de inteiros informando quais são os postos principais e artificiais;
- A – ponteiro para *double* contendo as M séries;
- C – ponteiro para *double* contendo os K grupos;
- cci – ponteiro para *double* contendo a inversa da matriz de covariância entre as vazões dos N postos.

- dmf – retorno da função *cuda_mahala()*. Trata-se de um ponteiro para *double* contendo as distâncias de cada série para cada grupo. Assim, *dmf* é uma matriz M x K onde suas linhas representam as séries, e as colunas os *clusters*, ou seja a posição *dmf[m, k]* representa a distância da série *m* ao grupo *k*.

```
extern "C" __host__ void cuda_mahala(int *nSeries, int *nClusters, int *nsis, int *stip,
                                     double *A, double *C, double *cci, double *dmf)
{
    //... Executa funcoes de inicializacao ....

    //************************* Aloca variaveis no device **********************
    int *stip_dev;
    double *A_dev, *C_dev, *cci_dev, *dmf_dev;

    checkCudaErrors(cudaMalloc((void ** ) &stip_dev, sizeStip));
    checkCudaErrors(cudaMalloc((void ** ) &A_dev, sizeA));
    checkCudaErrors(cudaMalloc((void ** ) &C_dev, sizeC));
    checkCudaErrors(cudaMalloc((void ** ) &cci_dev, sizeCci));
    checkCudaErrors(cudaMalloc((void ** ) &dmf_dev, sizeDmf));

    //************************* Copia variaveis pro device **********************
    checkCudaErrors(cudaMemcpy(stip_dev, stip, sizeStip, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(A_dev, matrizA, sizeA, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(C_dev, matrizC, sizeC, cudaMemcpyHostToDevice));
    checkCudaErrors(cudaMemcpy(cci_dev, matrizCci, sizeCci, cudaMemcpyHostToDevice));

    //************************* Chama o kernel *********************************
    int nThreads = 1024;
    int nBlocos = (*nSeries) / nThreads;
    mahala <200> <<<nBlocos, nThreads>>>( *nClusters, *nsis, stip_dev,
                                          A_dev, C_dev, cci_dev, dmf_dev);

    //******************* Copia a matriz Dmf do device pro host ******************
    checkCudaErrors(cudaMemcpy(matrizDmf, dmf_dev, sizeDmf, cudaMemcpyDeviceToHost));

    checkCudaErrors(cudaFree(stip_dev));
    checkCudaErrors(cudaFree(A_dev));
    checkCudaErrors(cudaFree(C_dev));
    checkCudaErrors(cudaFree(cci_dev));
    checkCudaErrors(cudaFree(dmf_dev));

    //... Executa funcoes de finalizacao ....
}
```

Figura 4-1: Parte do código fonte da função cuda_mahala()

A rotina *cuda_mahala()* é responsável por executar os seguintes passos:

- Alocar as estruturas de dados na memória do *device*;
- Copiar os dados para a memória RAM do *host* para a memória RAM do *device*. É na memória do *device* que as *CUDA-threads* irão atuar.
- Executar no *device* o *kernel* para cálculo das distâncias;

- Copiar os resultados da memória do *device* para a memória do *host*. Apesar de alocar e copiar os resultados para a memória do *device*, as funções do *host* não possuem acesso aos endereços de memória do *device*.
- Desalocar as estruturas no *device*.

A Figura 4-1 apresenta a parte do código fonte dos passos descritos acima.

No início da *cuda_mahala(),* são executadas algumas funções de inicialização. As matrizes *A, C e cci* são transformadas do formato *colunm major 1-based index* do Fortran para o formato *row major 0-based index* em C/C++ e armazenadas nas matrizes *matrizA, matrizC, matrizCci*. Para utilização do device, as variáveis estão declaradas com o sufixo "_dev", para facilitar a localização de onde cada estrutura está sendo utilizadas.

As matrizes *A_dev, C_dev, cci_dev, stip_dev e dmf_dev* são alocadas no *device* através da função ***cudaMalloc()*** e copiadas através da função ***mudaMemcpy()***, cujo sentido é definido pelo parâmetro ***cudaMemcpyHostToDevice***, indicando que o sentido da cópia é da memória da CPU para a memória da GPU. Em seguida o *kernel* ***mahala()*** é chamado para cálculo paralelo das distâncias na GPU, que será executados por ***nBlocos*** de ***nThreads*** cada. O resultado será armazenado na matriz *dmf_dev*. Após a execução do *kernel*, a matriz *dmf_dev* é copiada da memória da GPU para a variável matrizDmf, alocada na memória da CPU através da função ***mudaMemcpy()***, com o sentido definido pelo parâmetro ***cudaMemcpyDeviceToHost***. Por último as variáveis são desalocadas da memória da GPU através da função ***cudaFree()*** e são realizadas as funções de finalização.

Para cada implementação o *kernel mahala()* é substituído pelos *kernels mahala_gm(), mahala_sh() e mahala_cublas().* O valor 200 passado em ***mahala <200>*** é a dimensao máxima dos postos necessária para alocação estática dos vetores no kernel.

Para avaliar o desempenho dos três *kernels*, primeiramente os *kernels* foram chamados pela *cuda_mahala()* para os cálculos da inicialização dos K centróides, com o intuito de avaliar o desempenho da GPU apenas nos cálculos das distâncias. Para avaliar a aceleração total, levando em consideração o *overhead* de alocação e cópia de dados entre *host* e *device* foram geradas três versões do modelo GEVAZP, uma para cada *kernel*: Gevazp_gm, Gevazp_sm e Gevazp_CUBLAS.

Para facilitar a visualização do algoritmo a ser paralelizado na GPU, o algoritmo foi dividido em três passos principais (Figura 4-2). As entradas são as matrizes de séries de vazões A, a inversa da matriz de covariância entre as vazões dos postos *cci*, e a matriz de *grupos* C. A saída é a matriz de distâncias entre as séries e os grupos (*Dm).* Serão apresentadas diferentes estratégias para realização desses passos no *device* com o objetivo de maximizar o desempenho através do acesso à memória e utilização dos *CUDA-threads*. São explorados conceitos como latência, coalescência, compartilhamento de memória, cooperação entre CUDA-*threads*, blocos de *threads*, ocupância além da utilização de algoritmos de redução paralela.

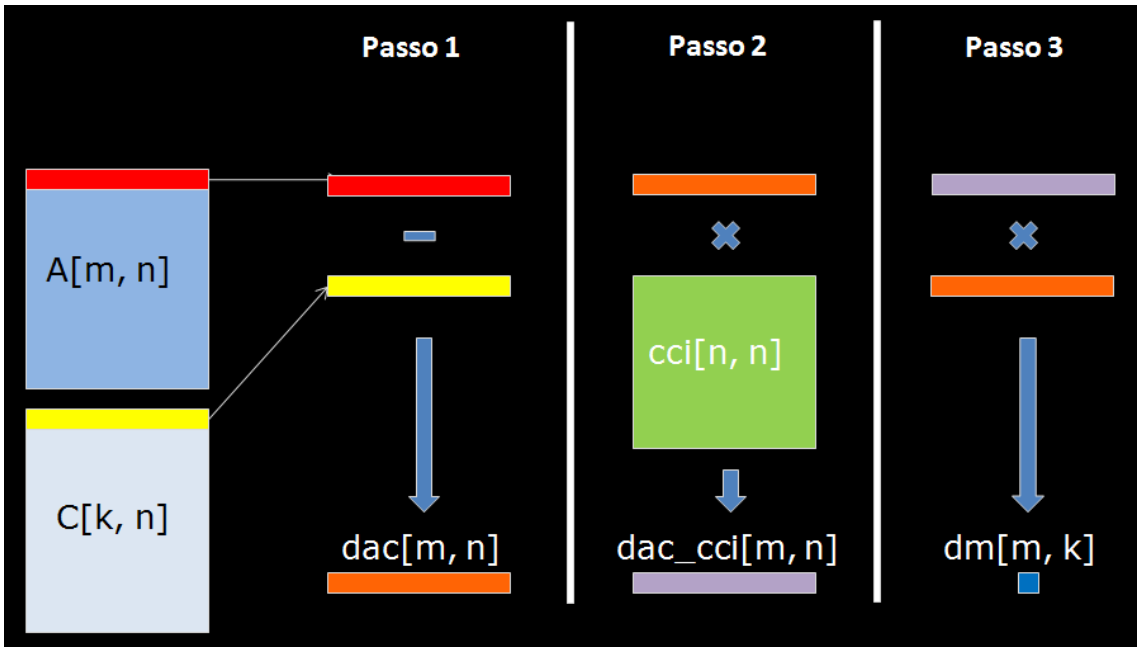


Figura 4-2: Passos para cálculo das distâncias de Mahalanobis

4.2 Gevazp_gm – Utilização de Memória Global

Nesta primeira versão, o objetivo foi traduzir para CUDA o código que calcula a distância de Mahalanobis entre uma vazão *m* e um grupo *k*, já existente em FORTRAN (na CPU). Essa implementação visa realizar uma avaliação inicial do potencial paralelo das GPUs. O domínio do problema foi subdivido de forma que cada *CUDA-thread* fosse responsável por uma, e apenas uma série de vazão, das M vazões existentes. Assim, cada *thread* realiza a tarefa de calcular as distâncias entre uma série *m* e os K *clusters*, retornando uma linha da matriz *dmf*.

Em resumo, cada *thread* irá realizar os seguintes passos:

- Calcular a diferença entre o a série ***m*** (vetor A[m,...]) e o grupo ***k*** (vetor C[k, ...]) e armazenar em um vetor, denominado ***dac*** (diferença entre os vetores de **A** e **C**).
- Multiplicar o vetor *dac* pela inversa da matriz de covariância ***cci***, e armazenar em um vetor ***dac_cci***.
- Calcular o produto interno entre o vetor ***dac*** e o vetor ***dac_cci***, obtendo a distância.
- Esse processo é repetido para todos os K grupos.

A Figura 4-3 apresenta a distribuição das *threads* enquanto a Figura 4-5 descreve o algoritmo executados por cada uma das *threads*.

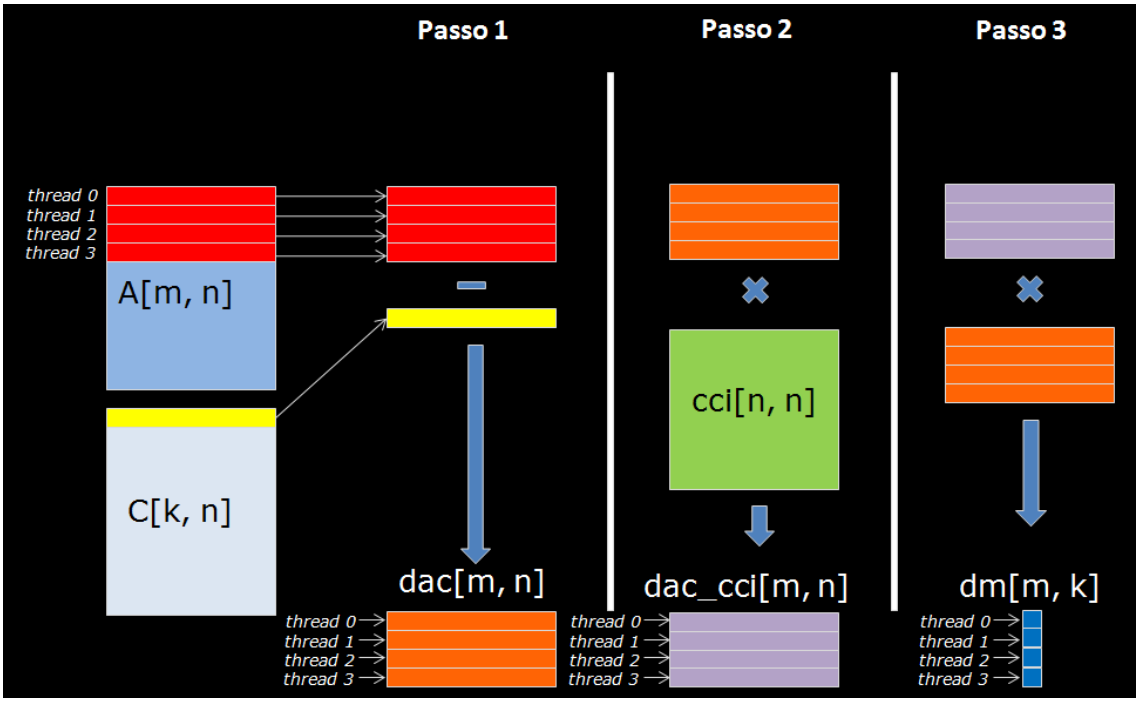


Figura 4-3: Distribuição das *threads* utilizando-se memória global

4.2.1 Definição de blocos e threads

Como as *threads* não vão cooperar entre si, o número de *threads* em cada bloco só é relevante no que diz respeito ao limite máximo, que atualmente é de 1024 *threads* por bloco. O número de vazões a serem agregadas no GEVAZP originalmente é igual a 1000. Apenas para facilitar os testes, o número de vazões foi passado para 1024 pelos seguintes motivos:

- Facilitar a definição dos tamanhos dos blocos, sendo múltiplos do tamanho dos *warps* e dos *half-harps*, que atualmente é de 32 e 16 *threads*, respectivamente;
- Facilitar a codificação dos *kernels*;
- Facilitar o balanceamento de carga entre as *threads*;
- 1000 objetos é o número utilizado hoje apenas por limitações de tempo na execução do modelo GEVAZP. Atualmente existem estudos no CEPEL variando esse número de objetos para uma melhor representação das séries no processo de agregação. No modelo NEWAVE atualmente são utilizados 100.000 objetos.

Assim, foram alocadas para esta execução, **1024 *threads* por bloco** distribuídas na **dimensão y**, e neste caso um *grid* de apenas 1 bloco (Figura 4-4).

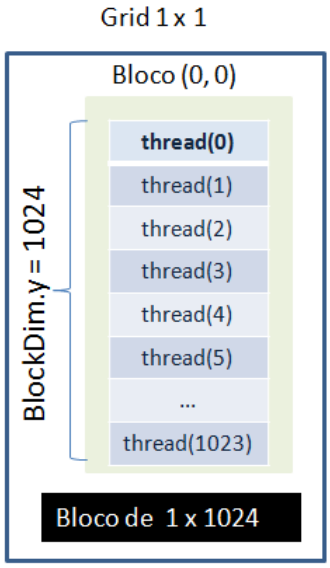


Figura 4-4: Definição do grid e blocos. Blocos de 1 x 1024 *threads*. *Grid* de 1 x 1 blocos

A implementação é composta por dois *kernels*:

- *__global__ mahala_gm();*
- *__device__ distanciaMahalanobis().*

A função ***mahala_gm()*** é chamada pelo *host* e é o ponto em que o controle da execução é passada da CPU para a GPU. Primeiramente define-se o identificador da *thread*, *threadId*, localizando a série na dimensão y de acordo com o número de blocos. Então, percorre-se um laço nos K grupos calculando o vetor *dac* e chamando o *kernel* ***distanciaMahalanobis()*** para realizar os cálculos descritos nos passos 2 e 3 do algoritmo 1 (Figura 4-5).

Algoritmo 1: Algoritmo utilizado no Gevazp_gm

```
idThread ← BlockIdx.y*BlockDim.y + threadIdx.y

for k = 0 to K                                   //Laço nos K grupos
   for j = 0 to N                                //Laço nos N índices dos vetores
      dac[ j ]  ← A[idThread] [ j ] – C[idThread] [ j ]        // Passo 1
   end for

   dac_cci ← multiplicacaoVetorMatrix (dac, cci)               // Passo 2

   dm [ idThread] [k] ← produtoInterno (dac, dac_cci)          // Passo 3


end for
```

Figura 4-5: Estrutura do cálculo das distâncias utilizando-se memória global

4.2.2 Otimizações do código fonte em CUDA

Quando se está desenvolvendo um código em CUDA para GPU, ou até mesmo portando para GPU um código fonte escrito originalmente para CPU, devido à sua arquitetura, deve-se ter em mente que o desempenho da aplicação é muito mais sensível na arquitetura GPU. Por exemplo, operações de adição e multiplicação são feitas em um ciclo de *clock* da GPU, enquanto operações de exponenciação ou divisão, por exemplo, são realizadas através de algoritmos que consomem dezenas ou até centenas de ciclos de *clock*. Além disso deve-se estar atento a estruturas condicionais, como por exemplo condições *if – then- else*. O maior impacto de uma estrutura condicional na execução de um *kernel*, é existir uma divergência do caminho percorrido na execução das *threads* dentro de um mesmo *warp*. Nesse caso, o *warp-scheduler* vai serializar a execução das diferentes *threads*, degradando o desempenho da aplicação.

4.2.2.1 Otimização do número de acessos à memória global

A Figura 4-6 apresenta o código do *kernel distanciaMahalanobis()*.

```
__device__ void distanciaMahalanobis(int nsis, int dimensao, int *stip,
{
    double dm;
    *dmf = 0;

    for (int i = 0; i < nsis; ++i) {
        if(stip[i] == 0) continue;
        dm = 0;

        for (int j = 0; j < nsis; ++j) {
            if(stip[j] == 0) continue;
            dm += dac[j]*(cci[i + j*dimensao]);
        }
        *dmf += dm*dac[i];
    }

    *dmf = arredonda(dmf);
}
```

1

2

Figura 4-6: Código fonte em CUDA do kernel distanciaMahalanobis()

No *trecho 1* em destaque, podemos identificar que é realizado uma operação condicional, *if(stip[i] == 0),* onde o código subsequente só será executado se o elemento do vetor *dac* representar um posto principal. O vetor ***stip*** armazena com 1 a posição do posto principal, e com 0 a posição do posto artificial. Inspecionando o código, verifica-se que não existe divergência entre *threads* uma vez que o vetor *stip* é o mesmo para todas as *threads*. Entretanto, como estamos usando memória global deve-se perguntar qual o custo para se realizar cada *if*, no código em destaque. Nos casos executados nos testes iniciais, existe um total de nsis = 112 postos, sendo que 58 são postos principais e 54 postos artificiais. São realizados 2 laços de 1 a 112. O *if* do laço mais externo é executado 112 vezes. Destas, 58 entram no laço mais interno, que executa o *if* também 112 vezes a cada entrada. Então o total de acessos à memória global são 112 + 58*112 = 6608 acessos, a cada distância calculada. Como são calculadas um total de M*K distâncias, temos um total de 6608*M*K acessos à memória global. Se considerarmos, por exemplo, o mês de Julho, que são utilizados K = 513 e M =1024 vazões, temos um total de **3.471.261.696** acessos à memória global.

Como os postos artificiais não contribuem para o cálculo das distâncias, modificaram-se as estruturas de dados utilizadas na função *cuda_mahala(),* para armazenar apenas as informações dos postos principais, ou seja, as dimensões dos vetores passaram de 112 para 58 e a matriz *cci* passou de 112x112 para 58x58 de acordo com o vetor *stip*. Com essa modificação, foi retirado do código as condições ***if(stip[i] == 0)***, reduzindo em 6608*M*K acessos à memória global.

4.2.2.2 Otimização de arredondamento

No *trecho 2* da Figura 4-6, o valor da distância é arredondado com uma determinada precisão em casas decimais. Nesse arredondamento, a distância é dividida por uma potência de 10, ou seja, 10^N. A princípio este código não nos parece interferir no desempenho. Entretanto, enquanto uma operação de multiplicação é realizada em 1 ciclo de *clock* na GPU, uma operação de divisão pode levar até 36 ciclos de *clock*. Assim, a terceira otimização no código foi substituir a operação de divisão (distância/10^N), por uma multiplicação por 10^{-N}.

Os testes iniciais de desempenho mostraram que as três otimizações diminuíram em aproximadamente **50%** o tempo de execução na GPU. Isto demonstra que ao desenvolver códigos em CUDA, a visão do programador deve ser muito mais crítica no que diz respeito aos detalhes da implementação em *hardware* de determinadas operações da GPU, e principalmente, como está sendo feito o acesso aos diferentes tipos de memória do *device*. Estas otimizações foram incorporadas nas versões descritas nas próximas seções.

4.3 Gevazp_sh – Utilização de Memória Compartilhada

A memória compartilhada está localizada dentro de cada *Stream Multiprocessor*, ou seja, *on-chip*, enquanto a memória global está localizada no *device*, ou seja, *off-chip*. Dessa forma, o acesso à memória compartilhada é muito mais rápido se comparado à memória global. Além disso, as *threads* pertencentes a um mesmo bloco cooperam entre si acessando os mesmos endereços da memória compartilhada. Na prática significa que endereços de memória lidos ou escritos por uma *thread* podem ser acessados pelas demais.

Infelizmente a quantidade de memória compartilhada é bem menor que a memória global. O SM possui apenas 48 KB de memória compartilhada enquanto a memória global da GPU pode chegar a até 24 GB. Assim, é necessário carregar blocos de dados da memória global para a memória compartilhada de no máximo 48 KB, que serão utilizadas pelas *threads* executadas em um mesmo SM. Baseado nessas características, vamos analisar o algoritmo anterior e estudar quais modificações podem ser feitas para melhorar seu desempenho.

Observando para o laço mais interno do *kernel distanciaMahalanobis()* (Figura 4-6) verifica-se que todas as *threads* acessam os mesmos valores da matriz *cci*. O dado que muda

entre as *threads* é o vetor *dac*, calculado no *kernel mahala_gm()*. A matriz é acessada $\boldsymbol{nsis^2}$ vezes a cada chamada do *kernel distanciaMahalanobis()*, por *thread*. Como cada *thread* é responsável por calcular as distâncias entre uma série e os K grupos a matriz *cci* é acessada $\boldsymbol{K*nsis^2}$**.** Considerando o mesmo exemplo da seção anterior, K = 513, M = 1024 e nsis = 58, temos um total de **1.732.702.208** acessos à matriz *cci* na memória global. Como os acessos à memória global são relativamente lentos, isso pode degradar o desempenho do *kernel*, deixando *threads* em espera por até centenas de ciclos de *clock*.

Para reduzir o acesso à memória global, é possível carregar blocos da matriz *cci* para a memória compartilhada. Como o código está trabalhando com dados do tipo *double* (8 *bytes*), o tamanho da matriz *cci* em *bytes* é de $nsis^2*8$ = 26.912 *bytes*, aproximadamente 26,3 KB. Assim a matriz *cci* pode ser completamente carregada na memória compartilhada.

No cálculo do vetor *dac* no *kernel mahala_gm()*, cada *thread* realiza ***2*nsis*** acessos à memória global (*A[m] – C[k]*), a cada grupo, ou seja, ***2*nsis*K*** acessos à memória. Uma maneira de esconder a latência de acesso à essa memória é o *warp scheduler* executar *múltiplos warps* por SM. Executar um número maior de *threads* por SM aumenta a probabilidade do *warp scheduler* encontrar *threads* prontas para serem executadas enquanto outras ainda estão buscando os dados na memória. Essa é a idéia principal do conceito de TLP (*Thread Level Parallelism)*. Aumentando-se o número de *threads*, aumenta-se o nível de ocupação do SM (seção 2.3.5) e possivelmente o desempenho do *kernel*.

Então o desafio é construir um algoritmo que seja capaz de utilizar memória compartilhada e um grande número de *threads*. Para isso modificou-se o algoritmo original de maneira que cada *thread* seja responsável pelo cálculo de apenas um índice do vetor de cenários de vazões, e não mais por cada vazão. A Figura 4-7 mostra o novo esquema:

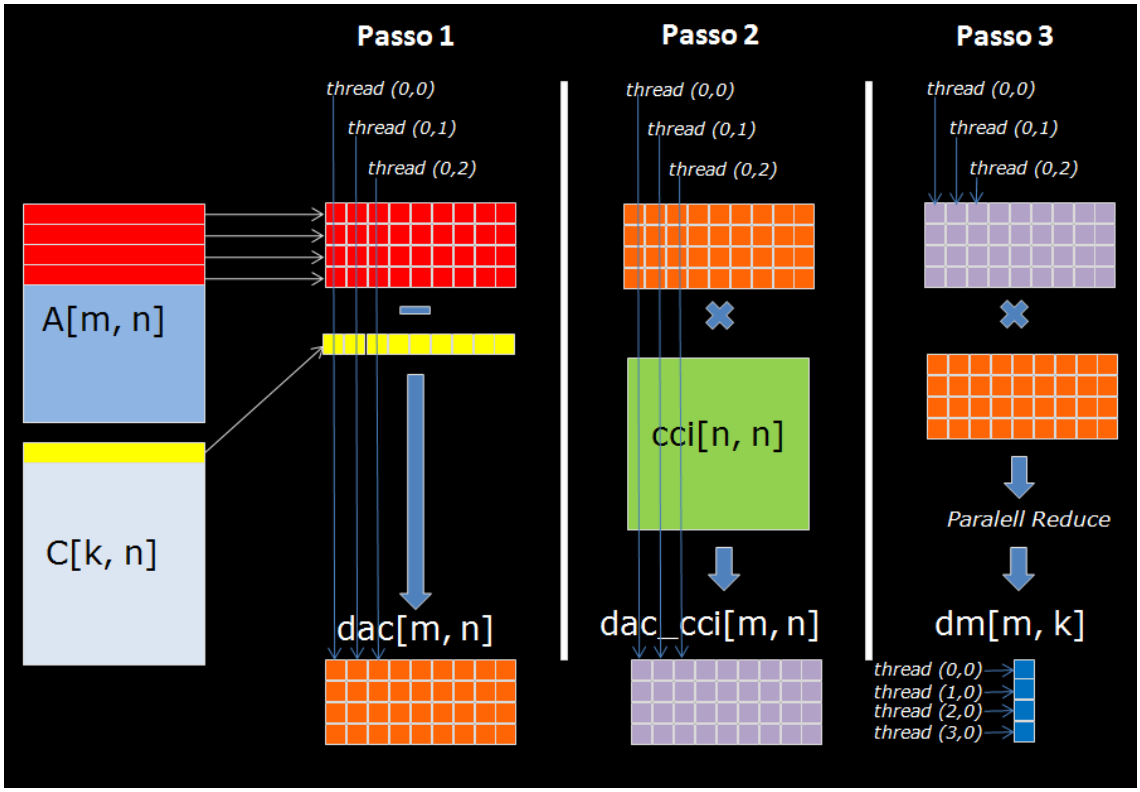


Figura 4-7: Distribuição das *threads* utilizando-se memória compartilhada

O passo 1 é responsável pelo cálculo do vetor *dac*. Alocando este vetor na memória compartilhada, cada uma das N *threads* calcula apenas um índice *n* do vetor *dac [n] = A[m, n] – C[k, n]*), onde N é a dimensão dos vetores. Como este cálculo é realizado para os K grupos, são realizados K*2 leituras da memória global e K escritas na memória compartilhada. Como o valor de *A[m, n]* é independente de K, carregamos seu conteúdo para um registrador antes do laço, otimizando K acessos à memória global.

No passo 2, cada *thread* é responsável pelo cálculo de um índice do vetor *dac_cci*, que é o resultado da multiplicação do vetor *dac* pela matriz *cci*. Como *dac* foi alocado na memória compartilhada, ele é reaproveitado por todas as N *threads* de uma linha. Para carregar a matriz *cci* na memória compartilhada, cada *thread* é responsável por carregar um ou mais elementos da memória global. Como *cci* tem dimensão N x N, cada *thread* será responsável por carregar um ou mais elementos da matriz, de acordo com a configuração do bloco de *threads*.

No passo 3 calcula-se a distância de Mahalanobis *dm[m, k]* através do produto interno entre os vetores *dac_cci* e *dac*. Nesta operação tem-se N *threads* que devem retornar apenas um resultado final *dm[m, k]*. Alocando-se o vetor *dac_cci* na memória compartilhada podemos realizar o que chamamos de *parallel reduce* ou *parallel reduction*. *Parallel*

reduction é uma técnica conhecida em computação paralela que reduz a complexidade *O(n)* de uma tarefa serial, para uma complexidade *O(log n)* utilizando **n** *threads*. A cada passo do algoritmo o número de *threads* "ativas" reduz pela metade, até restar apenas uma. O acesso à memória compartilhada nesta versão é otimizada pois realiza o acesso coalescente à memória, ou seja, as *threads* dentro de um mesmo *warp* acessam endereços sequenciais de memória.

Esta técnica mostrada na Figura 4-8 foi utilizada para o cálculo de *dm[m, k]*.

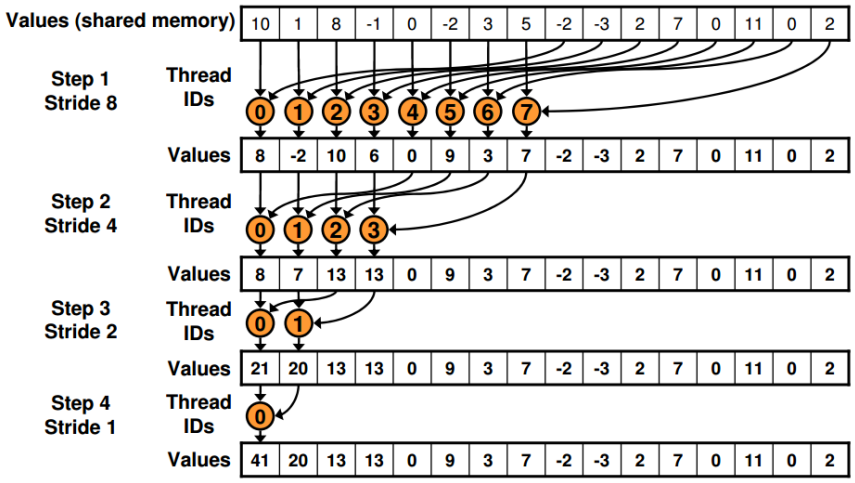


Figura 4-8: *Parallel Reduction* em memória compartilhada [39]

A Figura 4-9 apresenta os passos descritos indicando a alocação das estruturas em cada tipo de memória, e o algoritmo completo é mostrado na Figura 4-11.

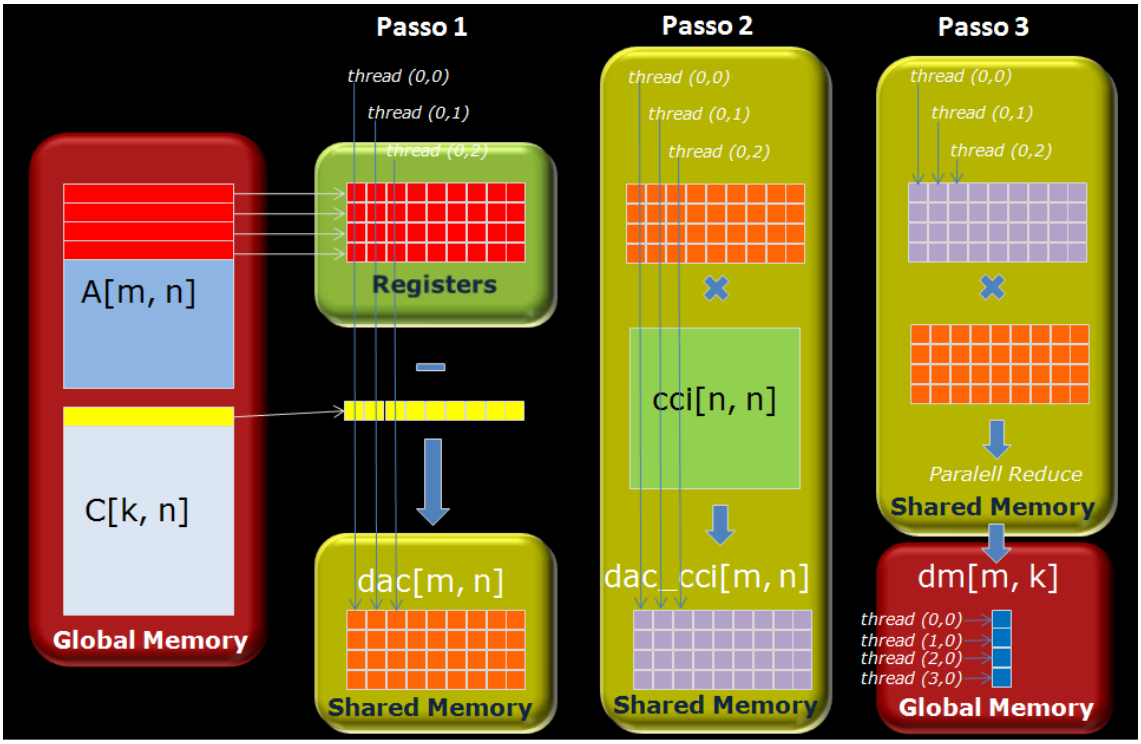


Figura 4-9: Alocação de memória das estruturas da versão Gevazp_sm

4.3.1 Definição de blocos e *threads*

Como as *threads* vão cooperar entre si, deve-se definir os blocos de *threads* de maneira a otimizar o acesso à memória compartilhada, que é limitada em 48 KB. Como as *threads* são executadas em múltiplos de 32 (*warp size)* e a dimensão N é dada pelo número de postos principais, que é igual a 58 nesta versão, definimos o número de *threads* igual a 64 na dimensão ***x*** do bloco. Assim, a matriz *cci* irá ocupar 64x64x8 = 32 KB, nos restando apenas 16 KB de memória compartilhada para os vetores *dac* e *dac_cci*. Estes vetores estão limitados em 8 KB cada. Definindo a dimensão y do bloco igual a 16, teremos 16x64x8 = 8 KB.

Dessa forma, para esta execução definimos blocos 2D (x,y) com dimensões 64 x 16, ou seja, 1024 *threads* por bloco. O número de blocos é dado por ***M / blocoDim.y***, que para M = 1024 é igual a 64 blocos.

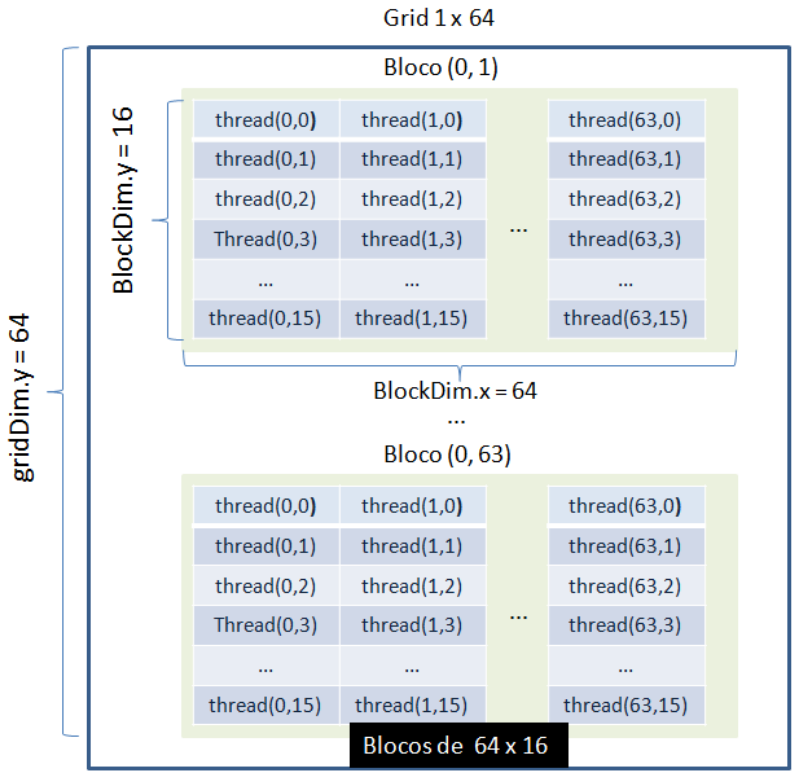


Figura 4-10: Definição de *grid* e blocos. Blocos de 64 x 16 *threads*. Grid de 1 x 64 blocos

4.3.1.1 Acesso coalescente à memória global

O acesso coalescente à memória é quando os dados são acessados em endereços contíguos na memória. Quando o acesso não é contíguo, o número de endereços na memória que são "pulados" no acesso é chamado de *stride*. Aceso não coalescente à memória global degrada o desempenho da aplicação afetando de dados que podem ser acessados por segundo.

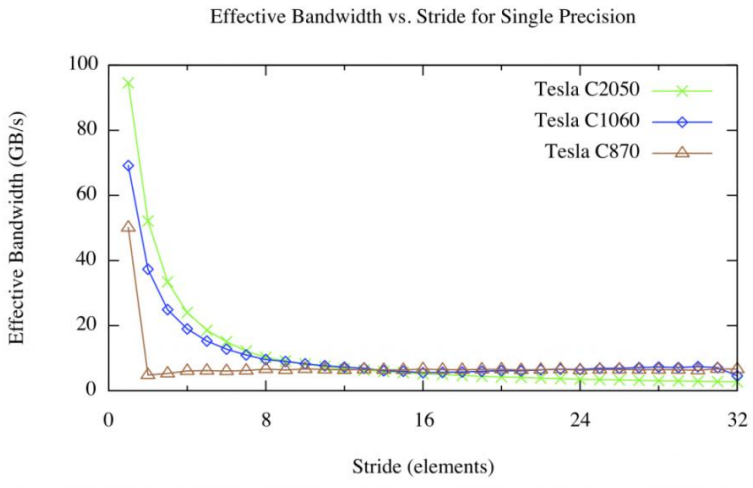


Figura 4-7: Impactos do *stride* na largura de banda da memória global [40]

No *trecho 1* da Figura 4-6, a matriz *cci* é acessada por colunas, ou seja, são feitos acessos com *stride* igual à *dimensao*. Como a matriz *cci* é uma matriz de correlação, ela possui a propriedade de ser uma matriz simétrica, ou seja, um elemento a_{ij} é igual ao elemento a_{ji}. O código foi modificado para fazer acesso coalescente à memória, acessando as linhas no lugar das colunas.

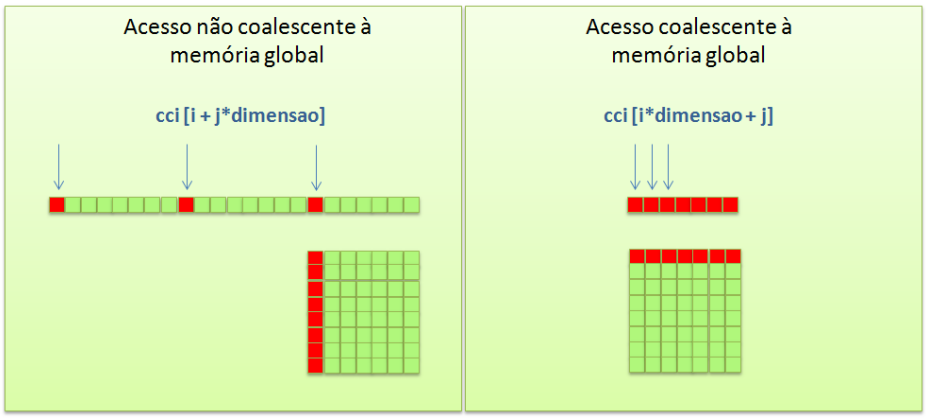


Figura 4-8: Acesso coalescente à matriz *cci*

O mesmo conceito foi aplicado ao acesso à matriz *cci* alocada na memória compartilhada.

Algoritmo 2: Algoritmo utilizado no Gevazp_sm

```
ty ← threadIdx.y                          //Identifica a thread dentro do bloco na dimensão y
tx ← threadIdx.x                          //Identifica a thread dentro do bloco na dimensão x

idSerie  ← BlockIdx.y*BlockDim.y + threadIdx.y   //Identifica a série na dimensão y
idPosto ← BlockIdx.x*BlockDim.x + threadIdx.x    //Identifica o posto na dimensão x
tx ←idPosto

Shared memory:   dac_sh [64][16]          //Aloca as estruturas na memória
Shared memory:   cci _sh [64][64]         //compartilhada utilizando os 48 KB
Shared memory:   dac_cci_sh [64][16]
Register: a, soma                         //aloca variáveis no registrador

for j = 0 to 3                            //Cada thread carrega 4 elementos da matriz
  cci _sh [ty + j*16][tx] ← cci [(ty + j*16) *64 + tx] //cci na memória compartilhada
end for

 syncthreads                              //Sincroniza as threads do bloco

a ← A[idSerie*64 + tx ]                   //Carrega o valor de A para o registrador

for k = 0 to K                            //Laço nos K grupos

    ____ Passo 1_____
    dac_sh [idSerie][tx] = a – C[k*64 + tx]   //Cada thread do bloco carrega um elemento tx
                                              //da série idSerie e do grupo k
    syncthreads                               //Sincroniza as threads do bloco


    ____ Passo 2_____
    for j = 0 to N                                  //Laço nos N índices dos vetores
       soma  ← soma + dac_sh [ ty][ j ]*cci_sh[tx][ j]   //Multiplicação do vetor ty pela matriz cci
                                                        //Acesso coalescente à memória cci_sh
    end for

    dac_cci_sh [ ty][ tx] = soma
    syncthreads                               //Sincroniza as threads do bloco

    ____ Passo 3_____
    parallelReduction_dot(dac_sh , dac _cci_sh)   //Calcula o produto interno dac x dac_cci
                                                  //através de parallel reduction utilizando
                                                  //a memória compartilhada
    if(ty = = 0)
       dm [ idThread] [k] ← dac_sh[0]             //A thread ty(0,idSerie) atualiza a distância

end for
```

Figura 4-11: Algoritmo do cálculo das distâncias utilizando-se memória compartilhada

4.4 Gevazp_cublas – Utilização da biblioteca CUBLAS

A biblioteca CUBLAS (*Compute Unified Basic Linear Algebra Subprograms)* [7] é uma versão paralela completa do padrão BLAS [41], capaz de fornecer até 17 vezes mais desempenho que as bibliotecas seriais em CPU. CUBLAS faz parte da *CUDA Runtime API* e implementa operações matriciais de três níveis: nível 1- vetor-vetor, nível 2 – vetor-matriz e nível 3 – matriz-matriz. Para utilizar a biblioteca, o desenvolvedor deve alocar espaço de memória para as matrizes e vetores no *device*, realizar as operações matriciais e recuperar os resultados. A biblioteca também fornece funções auxiliares para alocar, escrever e ler dados do *device*, além de definir em tempo de execução, qual a configuração ótima para os grids e blocos de *thread*, abstraindo estes detalhes da implementação. As funções em CUBLAS utilizam por padrão um armazenamento do tipo *columm-major* e índices *1-based*. Quando se está trabalhando com C/C++, que utiliza o armazenamento do tipo *row-major* e índices *0-base*, é importante considerar esta configuração a fim de se manter a compatibilidade dos *kernels* e reprodutibilidade de resultados.

Podemos utilizar a biblioteca nos passos 2 e 3 do algoritmo, já que a mesma não fornece uma função para subtração de vetores.

No passo 2, pode-se utilizar a função ***gemv()*** a qual realiza em paralelo a multiplicação de vetor por matriz. Esta solução iria chamar **M*K** vezes a função ***gemv()***, para realizar a multiplicação do vetor *dac* e da matriz *cci*. Em testes iniciais, verificou-se um desempenho bem abaixo das expectativas nas M*K multiplicações vetor-matriz. Isto se deve ao fato de que o desempenho da biblioteca CUBLAS aumenta com a dimensões de vetores e matrizes. Com N = 58, não foi possível extrair um bom desempenho da biblioteca.

Buscando otimizar o uso de CUBLAS no passo 2, o processo de se multiplicar M vetores *dac* pela matriz *cci* pôde ser mapeado em 2 passos:

- Construção de uma matriz *(matriz_dac)*, cujas linhas são os M vetores *dac*;
- Multiplicação da matriz *matriz_dac* pela matriz *cci*, através da função ***gemm().***

O resultado desta multiplicação é uma matriz cujas linhas são os vetores *dac_cci* de cada operação vetor-matriz que seria realizada Figura 4-12. Esta abordagem é mais apropriada para o domínio do problema, realizando K multiplicações da matriz m*atriz_dac* (1024 x 64) pela matriz *cci* (64 x 64).

Para o passo 1 e o passo 3 foram criados 2 *kernels*, o primeiro para obtenção dos vetores *dac* e o segundo utiliza a mesma técnica do de *parallel reduction*, descrita na seção anterior. A configuração de blocos e *threads* também foi mantida, exceto no passo 2, cujo tamanho do bloco é determinado pela biblioteca CUBLAS em tempo de execução. A primeira execução da função ***gemm()*** demora mais tempo que as demais. Isto se deve ao fato de que CUBLAS executa um *benchmark* que avalia diferentes tamanhos de blocos de *threads* para determinar aquele que fornece o melhor desempenho de acordo com a arquitetura e configuração da GPU que está sendo utilizada. Normalmente antes de realizar um laço com M chamadas à função ***gemm()***, executa-se uma chamada em dados quaisquer de mesmo tamanho para configurar as execuções seguintes (conhecido como *warm-up*).

A Figura 4-12 apresenta o esquema utilizando a biblioteca CUBLAS:

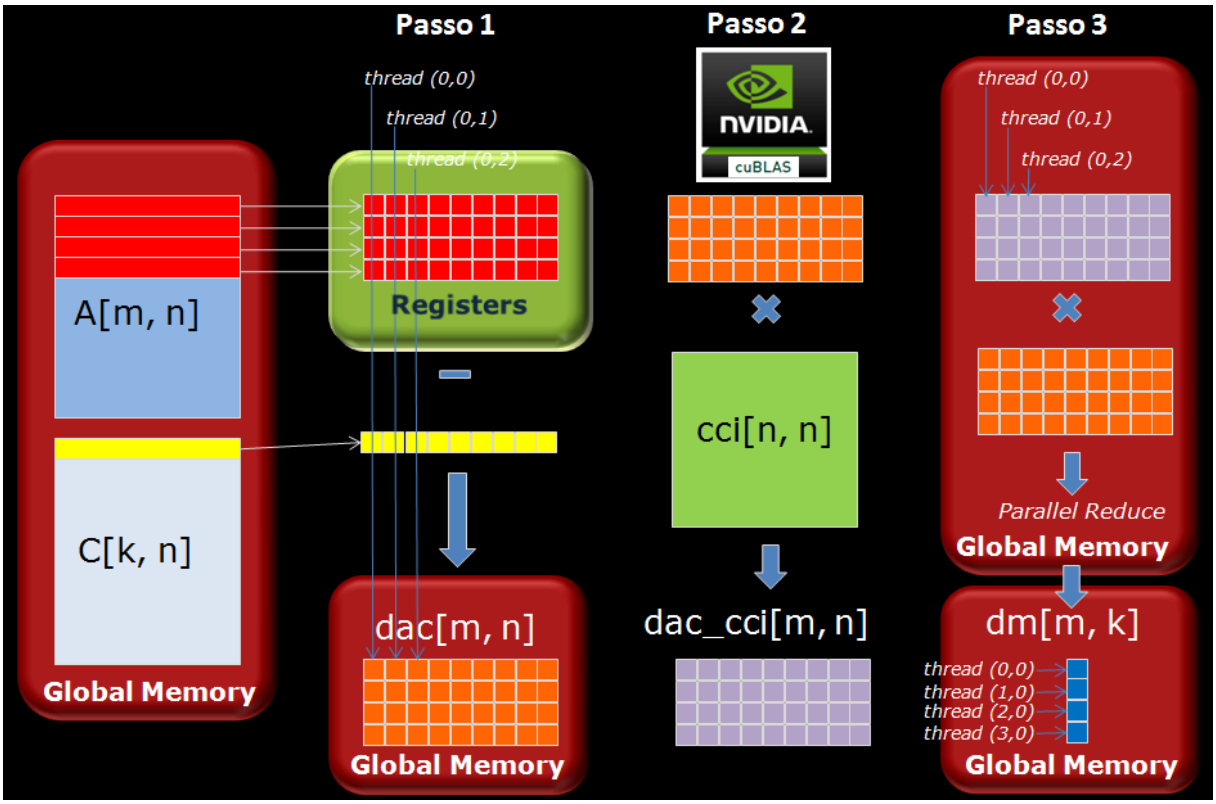


Figura 4-12: Alocação de memória e utilização da biblioteca CUBLAS - Gevazp_cublas

4.5 Implementação da solução paralela utilizando MPI

A segunda estratégia de paralelização explora o fato de que as aberturas dos cenários de afluência em um mesmo período são independentes, podendo ser geradas em paralelo. Nesta abordagem, propôs-se a utilização de um *cluster* de computadores. A geração de séries é dividida em vários nós interligados por uma rede, composto por um ou mais processadores, onde cada processador é responsável por gerar uma fração das K séries do período, reduzindo o tempo total do processo. Após a geração dos K cenários de um determinado período, as informações geradas em cada um dos processadores têm de ser consolidadas por um nó mestre, e enviadas a todos os processadores para serem utilizadas na geração do próximo período. Todo esse processo é coordenado através de troca de mensagens de sincronismo e de troca de dados entre os processadores, utilizando a biblioteca de troca de mensagens MPI. A abaixo ilustra a divisão das tarefas entre os processadores da rede e a redução de tempo associada.

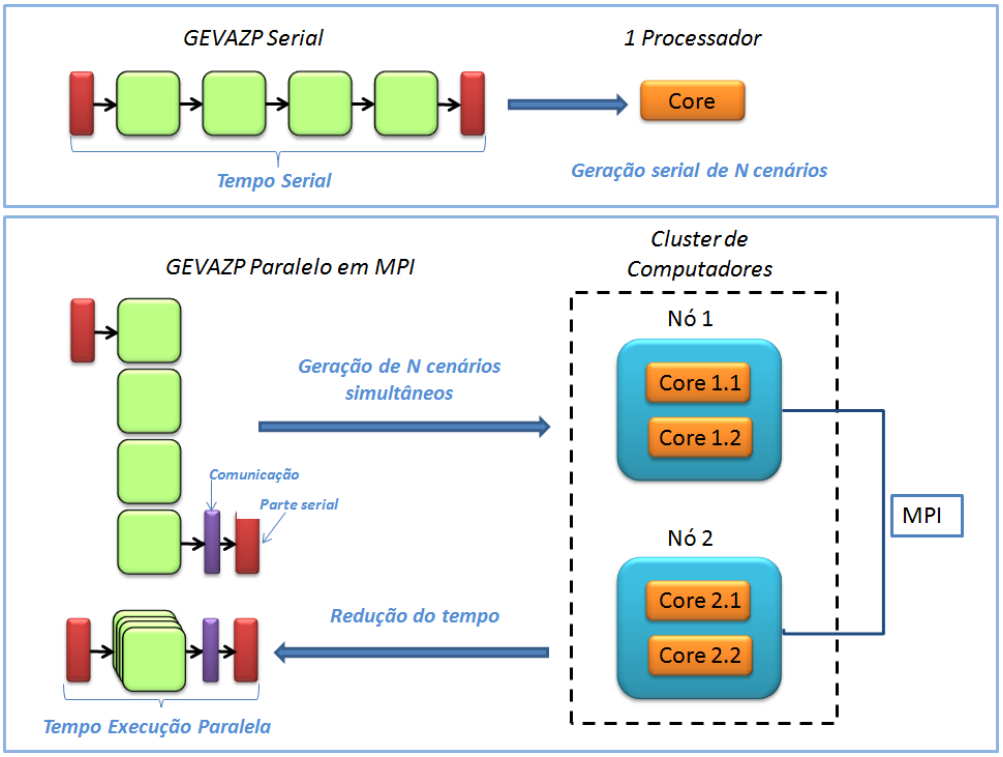


Figura 4-13: Geração paralela de cenários em MPI

Na Figura 4-13 os elementos em vermelho representam as tarefas seriais como leitura de arquivos e inicialização das estruturas de dados, e a totalização das séries e geração de arquivos ao final da execução. Em verde, são representadas a geração de cenários, onde está incluída o processo de agregação e, o *overhead* de comunicação está representado em roxo.

Na rotina principal do programa, é feita a inicialização do ambiente MPI. Os processadores recebem um identificador (*rank)* e o número de processadores que estão participando da execução. O processador *mestre* recebe *rank* = 0 e é ele o responsável por enviar dados aos processadores *escravos* e coordenar os pontos de sincronismo. Os processadores escravos não participam da leitura de arquivos e inicialização, por questões de sincronismo na leitura de arquivos. Eles participam apenas da geração das séries, ficando esperando em uma barreira o nó *mestre* enviar os dados para início da geração.

É criado um ponto de sincronismo na rotina de geração através da função da ***bcast()***, da biblioteca MPI, onde o nó mestre envia a todos os processadores os dados envolvidos na geração, assim como as informações de quais séries cada processador irá gerar. Após a geração de suas séries, os processadores escravos enviam sua parte para o processador mestre, através das rotinas ***send()*** e ***recv()***, que consolida as séries geradas naquele período, re-eviando para todos os processadores, os dados consolidados para serem utilizados no próximo período.

Após o processo de geração terminar, o processador mestre gera então os arquivos de saída do modelo como, por exemplo, o arquivo vazões, enquanto os processadores retornam à rotina inicial, onde ficam aguardando em uma barreira a chegada do processador mestre, para que todos os processadores finalizem o ambiente MPI e o programa juntos.

4.6 Implementação da solução paralela híbrida utilizando MPI e CUDA

Como o objetivo principal deste trabalho é a pesquisa do potencial de processamento das GPUs, as duas soluções apresentadas em MPI e CUDA foram integradas possibilitando executar o modelo em um *cluster* de GPUs. Esta solução se beneficia do ganho obtido em CUDA no processo de agregação de cenários e da resolução paralela da geração de cenários em MPI. A figura abaixo apresenta a solução híbrida em *cluster* de GPUs:

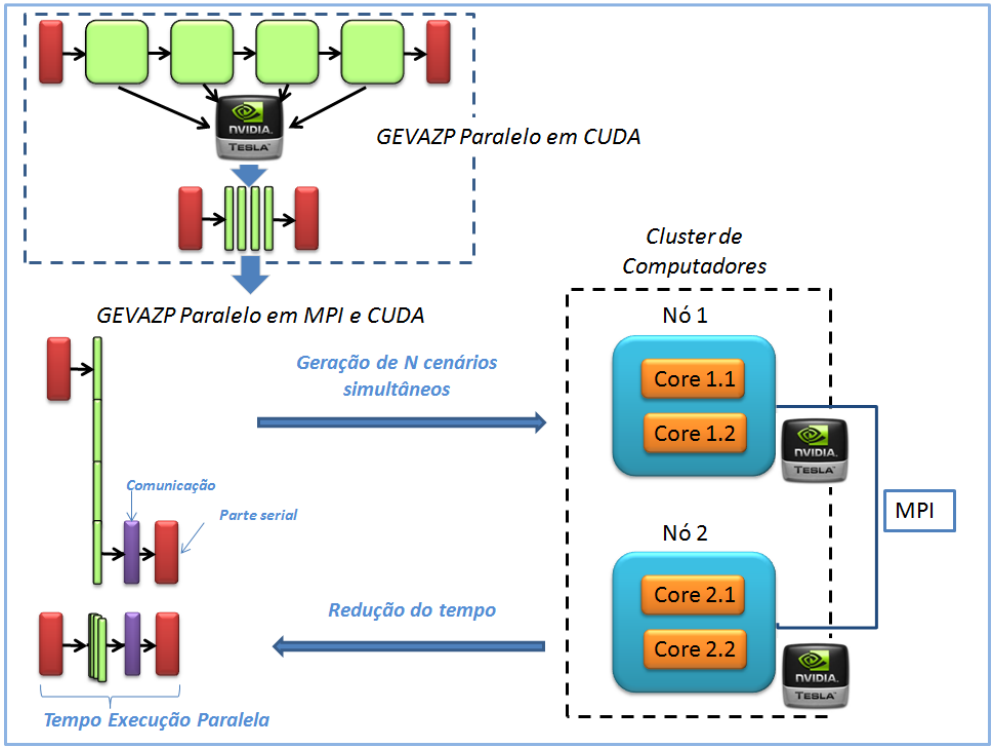


Figura 4-14: Cluster de GPUs: geração paralela de cenários em MPI e agregação de cenários em CUDA

Como não se dispunha de um *cluster* de GPUs, para realizar os testes com essa versão híbrida utilizando apenas uma GPU, nessa versão foi acrescentada a utilização de *CUDA-Streams* juntamente com a tecnologia Hyper-Q, introduzida na arquitetura Kepler, que permite que até 32 processadores façam acesso simultâneo a uma mesma GPU. Um *CUDA-Stream* representa uma sequência de instruções a serem executas na GPU, completamente independente das demais. Ao executar um *kernel* na GPU ele roda sob o *Stream* padrão. Para se executar um *kernel* em um *Stream* específico, primeiro deve-se criar o *Stream* no código em CUDA, e então passá-lo como o quarto argumento na chamada do *kernel:*

Ex: mahala_sh<<<nBlocos, nThreads, 0, Stream_1>>>

Através dos *Streams* em CUDA, é possível esconder latências de cópia de dados entre *host* e *device*, através de chamadas assíncronas da função ***memcpy()***. Quanto à execução dos *kernels*, é possível executar até 32 *kernels* em *Streams* paralelos na GPU, possibilitando principalmente diminuir o tempo ocioso da GPU. Na Figura 4-15, cada *core* da CPU cria um *CUDA-Stream* para a agregação de cenários e gera suas séries em paralelo com os demais processadores.

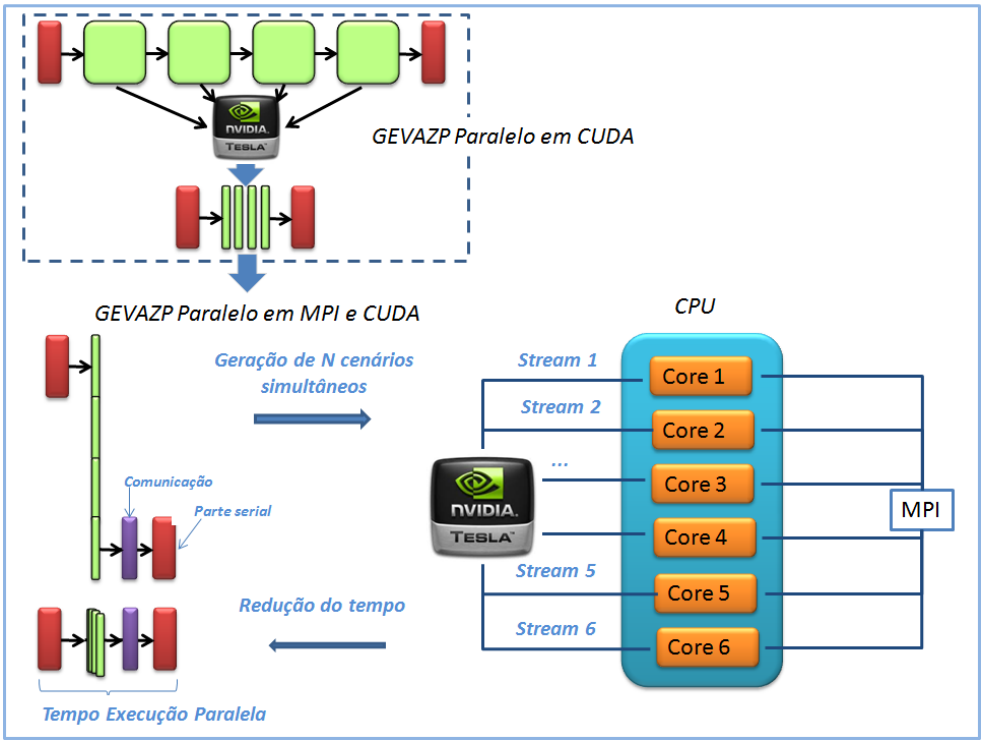


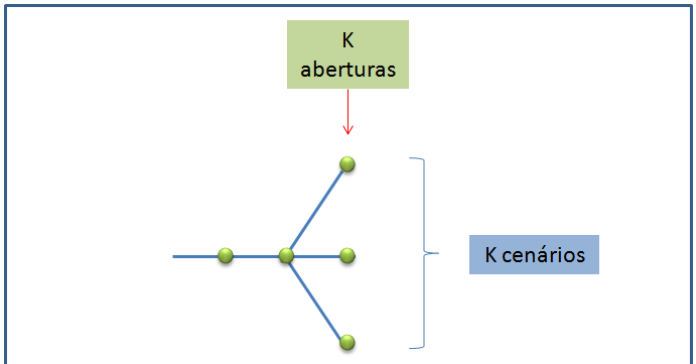
Figura 4-15: Utilização de *CUDA-Streams* na solução híbrida MPI e CUDA

5 EXPERIMENTOS E RESULTADOS

Os experimentos realizados têm o objetivo de investigar os ganhos de desempenho obtidos com a utilização de GPUs no modelo GEVAZP, assim como possíveis limitações na utilização dessa tecnologia.

Os testes foram realizados em uma GPU QUADRO K600, com 1 *Stream Multiprocessor* de 192 CUDA-cores de 875 MHz cada, 1 GB de memória RAM e 48 KB de memória compartilhada. Seu desempenho máximo teórico é de 336 GFLOPs em operações de ponto flutuante com precisão simples – 4 bytes (*float ou REAL*4*). Os testes seriais foram realizados utilizando uma CPU com 6 processadores Intel(R) Xeon(R), 2,00 GHz e 16 GB de RAM. Os códigos em CUDA foram desenvolvidos no sistema operacional Linux, distribuição CentOS 6.5, na IDE *NSight Eclipse Edition 7.0* compilados com *nvcc 7.0.*

Os casos utilizados foram os PMOs de Janeiro a Dezembro de 2008, compatível com a versão 5.0 do modelo GEVAZP. A Figura 5-1 mostra a árvore de cenários dos casos, onde K representa o número de agrupamentos de cada mês:



Mês	Jan	Fev	Mar	Abr	Mai	Jun	Jul	Ago	Set	Out	Nov	Dez
K	136	116	143	143	193	267	513	353	303	251	228	152

Figura 5-1: Árvore de cenários dos 12 PMOs

Foram medidos os tempos da versão serial, e das três versões paralelas: Gevazp_gm (memória global), Gevazp_sh (memória compartilhada) e Gevazp_CUBLAS (utiliza a biblioteca CUBLAS). Na metodologia utilizada, foram medidos os seguintes tempos:

- T1 - Tempo de inicialização dos K centróides;
- T2 - Tempo gasto pela rotina OPTRA;
- T3 - Tempo gasto no processo de agregação;

- T4 - Tempo total de execução do modelo.

O tempo de inicialização e o tempo da primeira iteração da rotina OPTRA são essenciais para avaliar o potencial de paralelização da GPU. Essas duas etapas calculam exatamente M*K distâncias, fornecendo o desempenho máximo que a GPU pode alcançar em relação à CPU. A partir da 2ª iteração, a rotina OPTRA sofre influência do conjunto de "grupos ativos", calculando um número menor que M*K.

Medir o tempo gasto no processo de agregação fornece um indicativo do *speedup* médio do processo iterativo OPTRA-QTRAN. Além disso, nessa medida não é considerado o tempo de pós-processamento das séries geradas. O tempo total de execução do modelo, conhecido como *overall speedup*, é o tempo gasto na agregação das séries em paralelo somado às partes seriais de leitura de arquivos, inicialização das estruturas de dados, e o pós-processamento da etapa de agregação das séries, como por exemplo, totalização das séries, testes estatísticos, escrita dos arquivos de vazões totais e dos arquivos de relatórios.

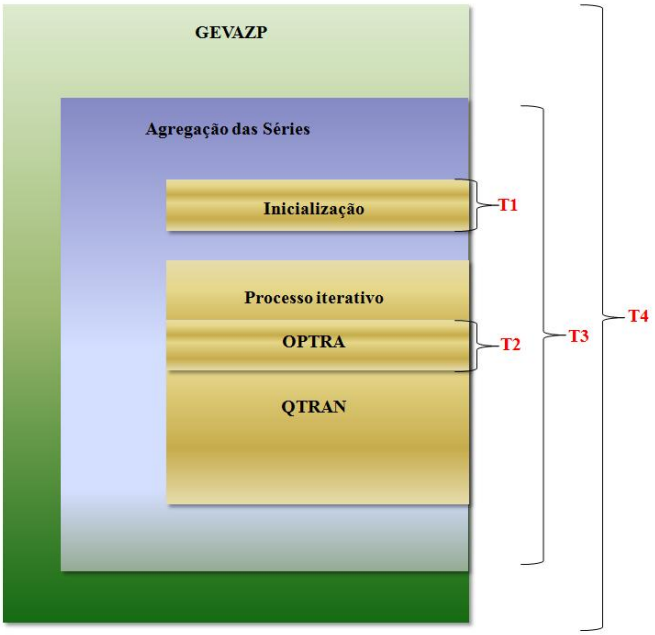


Figura 5-2. Tomada de tempos para avaliação do desempenho

De posse desses tempos é possível avaliar o desempenho das três versões paralelas em cada uma das etapas envolvidas, comparando-as com a versão serial modificada (versão_2).

5.1 Experimentos da etapa de inicialização

Os primeiros testes foram realizados na rotina de inicialização. Neste etapa a GPU calcula as distâncias das M séries para todos os K grupos, e então as M séries são alocadas nos grupos de maior semelhança. A Tabela 5-1 e Tabela 5-2 apresentam os tempos de inicialização para a versão serial (versão_2) e para cada uma das versões paralelas na GPU assim como as acelerações obtidas. Nestas tabelas, estão destacadas em verde a versão com menores tempos e consequentemente melhor desempenho.

A Figura 5-3, a Figura 5-6 e a Figura 5-9 apresentam os gráficos comparando os tempos seriais e paralelos das três versões. A Figura 5-4,

Figura **5-7** e Figura 5-10 mostram em destaque os tempos das versões em *double* e em *float* e a Figura 5-5, Figura 5-8 e Figura 5-11 apresentam o desempenho obtido.

Tabela 5-1. Resultados obtidos na medida dos tempos de inicialização

	Serial Tempo (s)	**Global Memory Tempo (s)**		**Shared Memory Tempo (s)**		**CUBLAS Tempo (s)**	
	versao_2	gm_double	gm_float	sh_double	sh_float	CUBLAS_d	CUBLAS_f
Jan	4,184	0,404	0,212	0,153	0,082	0,421	0,299
Fev	3,546	0,346	0,187	0,137	0,074	0,398	0,275
Mar	4,406	0,418	0,221	0,159	0,083	0,432	0,299
Abr	4,368	0,418	0,221	0,159	0,090	0,425	0,285
Mai	6,119	0,553	0,284	0,199	0,097	0,510	0,345
Jun	8,181	0,748	0,374	0,259	0,117	0,638	0,411
Jul	15,897	1,382	0,680	0,464	0,188	1,068	0,631
Ago	10,789	0,969	0,485	0,329	0,143	0,785	0,471
Set	9,247	0,840	0,422	0,288	0,129	0,683	0,444
Out	7,902	0,722	0,370	0,253	0,121	0,624	0,405
Nov	6,950	0,644	0,331	0,227	0,111	0,571	0,378
Dez	4,639	0,440	0,234	0,165	0,087	0,487	0,305

Tabela 5-2: Resultados obtidos dos speedups da etapa de inicialização

	Global Memory Speedup		**Shared Memory Speedup**		**CUBLAS Speedup**	
	gm_double	gm_float	sh_double	**sh_float**	CUBLAS_do	CUBLAS_fl
Jan	10,4	19,8	27,3	51,1	10,0	14,0
Fev	10,2	19,0	25,9	47,9	8,9	12,9
Mar	10,5	19,9	27,7	53,3	10,2	14,7
Abr	10,5	19,8	27,5	48,6	10,3	15,3
Mai	11,1	21,6	30,7	63,2	12,0	17,7
Jun	10,9	21,9	31,6	69,7	12,8	19,9
Jul	11,5	23,4	34,3	**84,5**	14,9	25,2
Ago	11,1	22,2	32,8	75,4	13,7	22,9
Set	11,0	21,9	32,1	71,8	13,5	20,8
Out	11,0	21,4	31,2	65,3	12,7	19,5
Nov	10,8	21,0	30,6	62,6	12,2	18,4
Dez	10,5	19,8	28,1	53,3	9,5	15,2

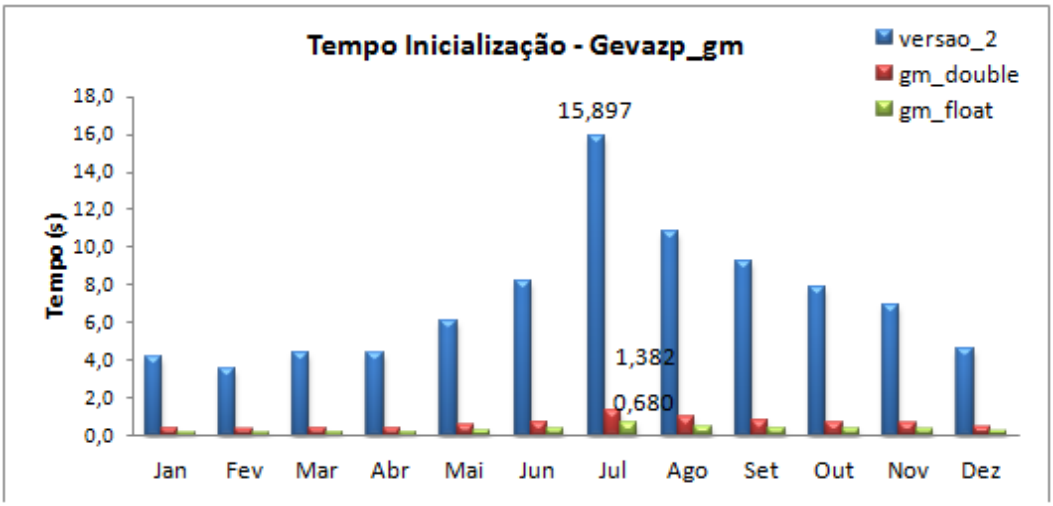


Figura 5-3: Comparação dos tempos da versão serial e da Gevazp_gm na inicialização

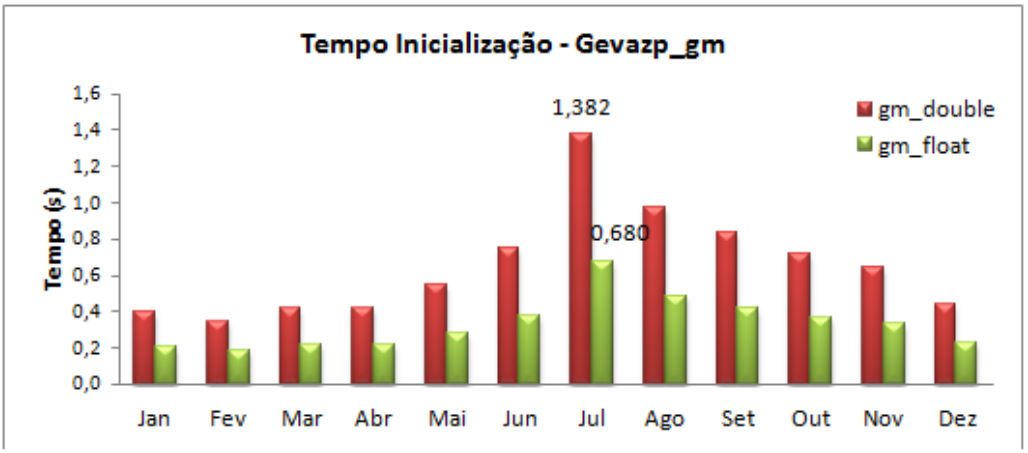


Figura 5-4: Detalhes dos tempos das versões *double* e *float* da Gevazp_gm na inicialização

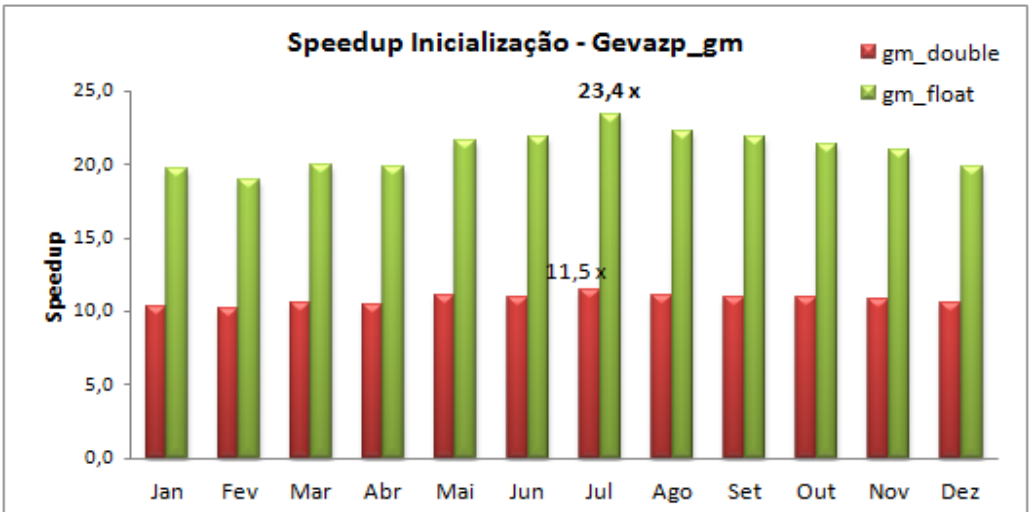


Figura 5-5: Speedup obtidos nas versões *double* e *float* da Gevazp_gm na inicialização

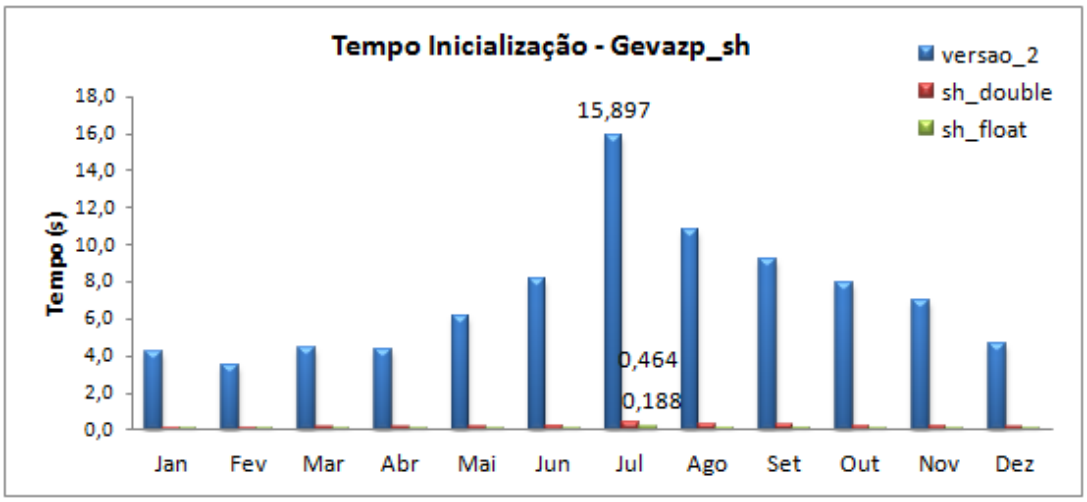


Figura 5-6: Comparação dos tempos da versão serial e da Gevazp_sh na inicialização

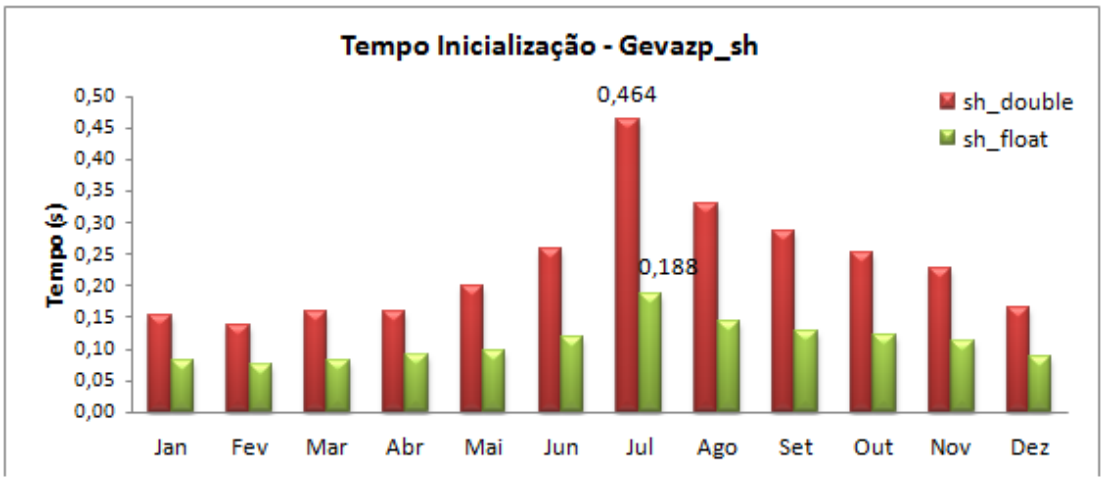


Figura 5-7: Detalhes dos tempos das versões *double* e *float* da Gevazp_sh na inicialização

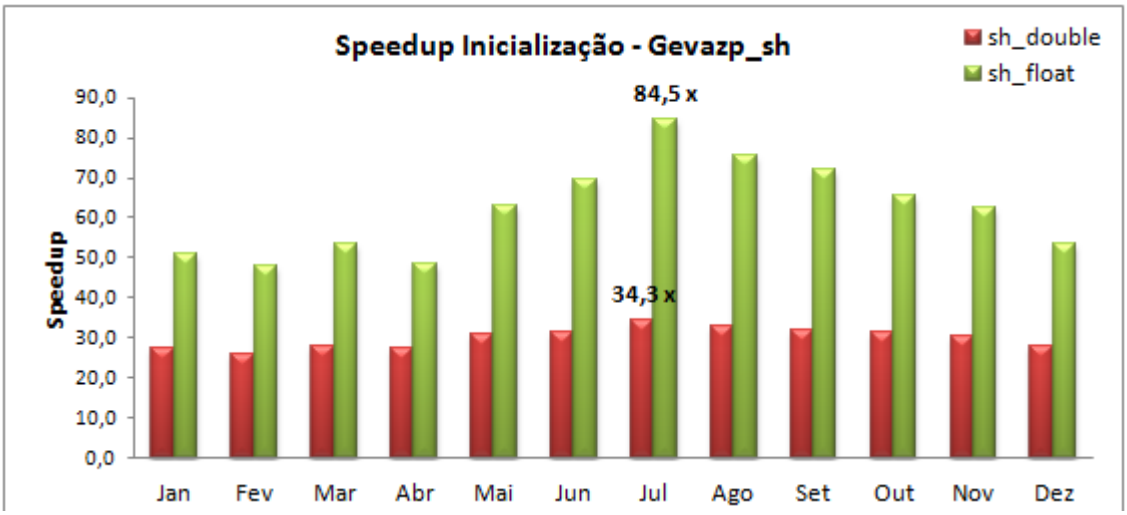


Figura 5-8: Speedup obtidos nas versões *double* e *float* da Gevazp_sh na inicialização

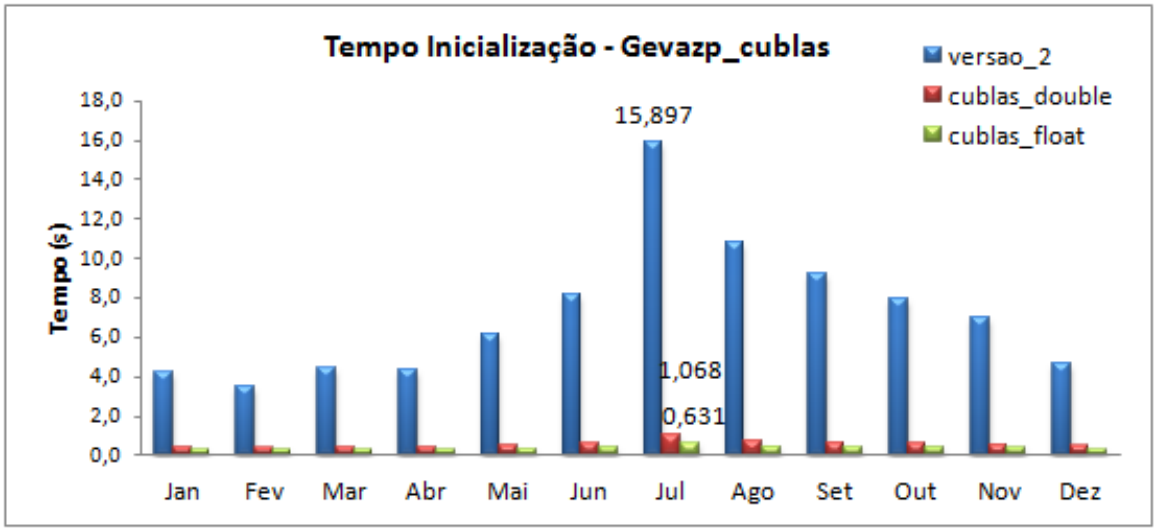


Figura 5-9: Comparação dos tempos da versão serial e da Gevazp_CUBLAS na inicialização

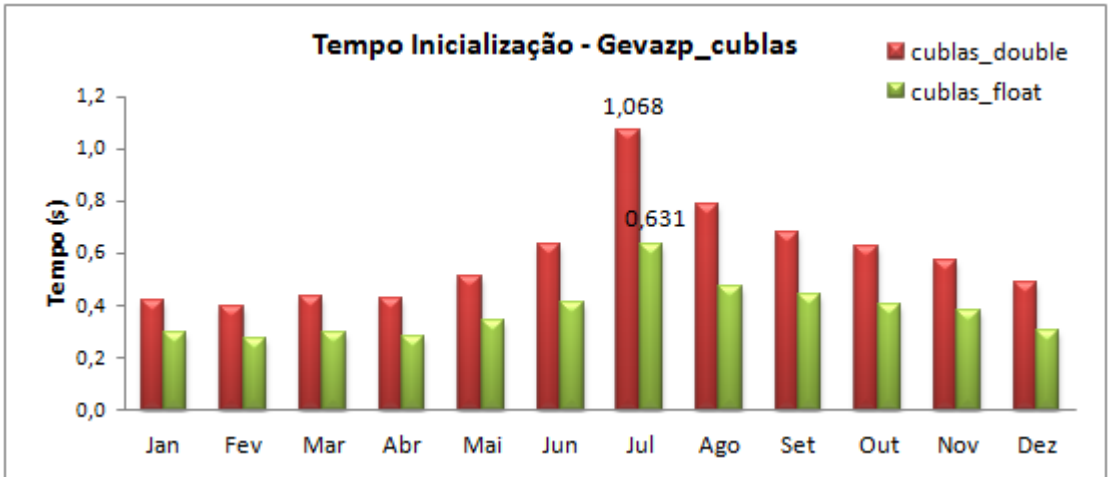


Figura 5-10: Detalhes dos tempos das versões *double* e *float* da Gevazp_CUBLAS na inicialização

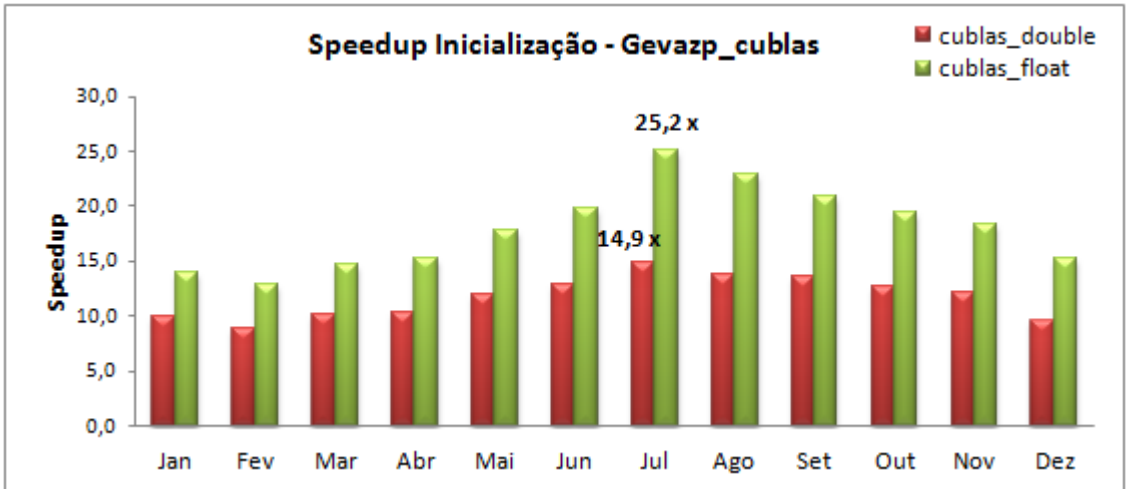


Figura 5-11: Speedup obtidos nas versões *double* e *float* da Gevazp_CUBLAS na inicialização

Inicialmente verificaram-se os valores para a versões utilizando precisão dupla (*double)* e precisão simples (*float*). Nas versões Gevazp_gm e Gevazp_sh observa-se da Tabela 5-1 que o tempo utilizando *float* cai a aproximadamente à metade comparado à versão utilizando *double*, em todos os casos rodados. Este fato fica visualmente mais claro através dos gráficos da Figura 5-3 e Figura 5-6. Já na versão Gevazp_cublas observa-se na Figura 5-10 que o tempo de execução em *float* pode chegar a aproximadamente 60% a 70% do tempo em *double*, indicando um desempenho não muito bom da biblioteca CUBLAS para cálculos em precisão dupla.

Estes dados indicam melhores desempenhos da GPU utilizando precisão simples. Esta diferença é devido ao fato de que são necessários dois acessos à memória para acessar os 8 bytes no caso de *double* enquanto para acessar 4 *bytes* em *float* é feito apenas um acesso à memória, e de que o SMx da GPU Quadro K600 possui 192 cores para cálculos em precisão simples e apenas 64 para precisão dupla. Uma vez que o arquivo de testes e de vazões apresentaram os mesmos valores para as versões nas duas precisões, conclui-se que a melhor escolha é utilizar *float* para os tipos de dados. Essa conclusão é fundamental já que na maioria das GPUs o desempenho teórico máximo em precisão simples é quase três vezes maior que em precisão dupla, permitindo assim utilizar o máximo potencial da GPU.

O próximo ponto a se observar é a redução de tempo obtida pelas três versões em relação à serial. A versão **Gevazp_gm** apresenta uma redução de tempo de cerca de **20 vezes** nos 12 casos, tendo seu melhor desempenho no mês de Julho, reduzindo o tempo de execução de **15,8 segundos para 0,680 s**, com uma redução de **23,4 vezes**, em destaque na Figura 5-5. Este resultado mostra que utilizando apenas a memória global somado às técnicas de otimização descritas na seção 4.2.2 foi possível obter ganhos consideráveis com esforço relativamente baixo para adaptar o código da CPU para GPU.

A versão Gevazp_sh foi a que apresentou maior redução dos tempos. De acordo com a Tabela 5-1**,** e Tabela 5-2, o desempenho variou de **47,9 vezes** para o mês de Fevereiro, apresentando um *speedup* máximo de **84,5 vezes** no mês de Julho, com uma redução de **15,8 segundos para 0,188 segundos** no tempo de execução, em destaque na Figura 5-8. Estes resultados mostram que a utilização otimizada da memória compartilhada em substituição à memória global melhorou significativamente o desempenho da aplicação, apresentando uma aceleração **3,6 vezes** maior do que na versão Gevazp_gm. É importante destacar que o algoritmo utilizado executa os três passos do cálculo das distâncias dentro do mesmo bloco de

threads, mantendo sempre que possível os dados em memória compartilhada. Esta abordagem foi essencial para obter o desempenho observado.

A versão **Gevazp_CUBLAS** por sua vez, apresentou redução de tempo que varia entre **12 e 25 vezes** nos casos rodados. Pode-se observar que seu melhor desempenho também é no mês de Julho, com redução de tempo de **15,8 segundos para 0,631,** o que equivale a uma redução de **25,2 vezes** em relação a serial (Figura 5-9 e Figura 5-11). Apesar da grande capacidade de processamento da biblioteca CUBLAS, seu desempenho nesta versão é limitado pela dimensão dos vetores (64 elementos). Seu baixo desempenho também se deve ao fato de que entre cada passo do algoritmo, é necessário escrever seus resultados na memória global para que o próximo passo possa ler essas informações.

De acordo com o exposto acima, nas próximas seções serão analisados os desempenhos dando um foco apenas para a versão Gevazp_sh utilizando *float*, já que esta mostrou o melhor desempenho dentre todas as versões.

5.2 Resultados da rotina OPTRA

Assim como na etapa de inicialização, a primeira iteração do processo OPTRA-QTRAN calcula as distâncias das M séries para todos os K grupos. O gráfico abaixo apresenta os *speedups* obtidos na primeira iteração para os 12 casos. Pode-se observar que o menor ganho foi de **85,6 vezes** no mês de Julho, e um o *speedup* máximo foi de **99,7 vezes** no mês Junho.

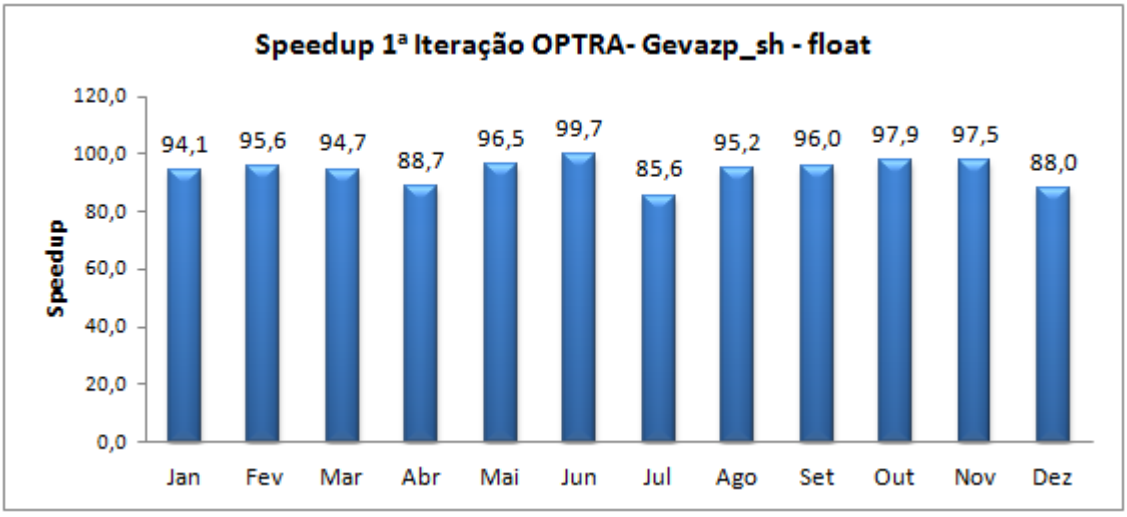


Figura 5-12: Speedup da primeira iteração da rotina OPTRA na versão Gevazp_sh

Estes resultados mostram que foi possível obter um ganho de quase **100 vezes** no cálculo das distâncias, indicando que é possível obter ganhos expressivos utilizando-se GPUs.

Apesar dos bons resultados apresentados, também é importante apontar as limitações encontradas durante o desenvolvimento deste trabalho. Estas limitações ficam claras ao acompanhar o processo iterativo de convergência OPTRA-QTRAN. Após a 1ª iteração OPTRA, o algoritmo classifica os agrupamentos que tiveram objetos transferidos como "grupos ativos". À medida que o processo vai convergindo, cada vez menos grupos estão ativos e o número de distâncias a serem calculadas diminui. Entretanto, na GPU não foi possível se beneficiar dessa característica do algoritmo, já que todas as *threads* de um mesmo bloco devem executar o mesmo conjunto de instruções. Em resumo, a cada iteração o tempo de execução na GPU permanece praticamente constante enquanto na CPU o tempo diminui. Como exemplo, a Figura 5-13 e Figura 5-14 apresentam os tempos e os *speedups* de cada iteração do processo de convergência do mês de Outubro. Observa-se desses gráficos o impacto da limitação citada, principalmente nas últimas iterações. O desempenho do processo de agregação é fortemente impactado pelo baixo desempenho das últimas iterações.

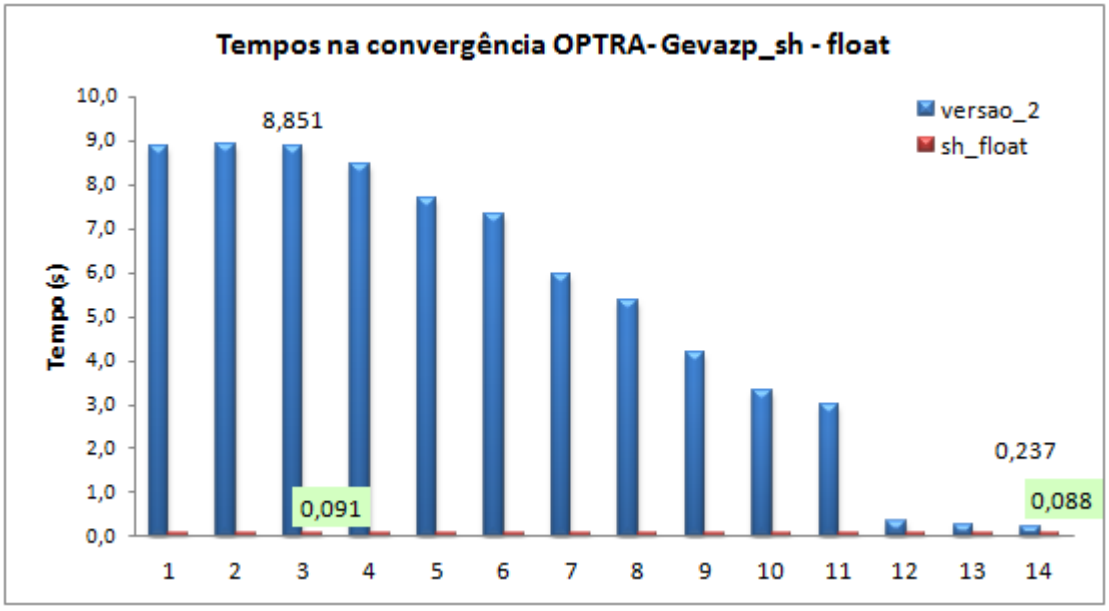


Figura 5-13: Resultados dos tempos da rotina OPTRA na convergência

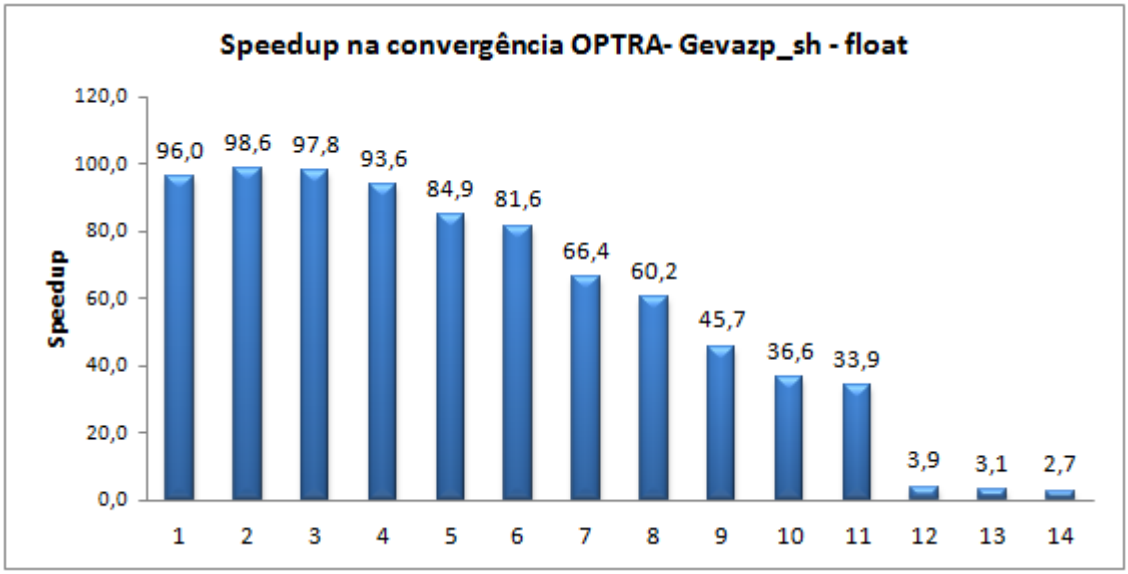


Figura 5-14: Resultados de Speedup na convergência da rotina OPTRA

Para os testes realizados, a rotina QTRAN não apresentou tempo computacional elevado a ponto de se beneficiar da paralelização, contribuindo com uma parcela serial no processo de agregação de cenários.

5.3 Resultados da agregação de cenários e overall speedup

Enfim são apresentados na Tabela 5-3 os resultados do processo de agregação de cenários e o tempo total de execução (*overall speedup*) do modelo GEVAZP paralelo em CUDA. A agregação obteve uma aceleração que varia de **18,5 vezes** no mês de Fevereiro, obtendo desempenho máximo de **38,1 vezes** no mês de Agosto. Como descrito anteriormente o desempenho desta etapa é impactada pela limitação da GPU não ser capaz de utilizar o benefício do algoritmo calculando apenas as distâncias dos "grupos ativos". Apesar disso, o processo apresentou um desempenho considerável reduzindo o tempo de agregação o de **84,2 para 2,2 segundos** no mês de Agosto (melhor caso).

Já na Tabela 5-4 podemos observar os tempos totais de execução e *overall speedup*. O acréscimo de tempo da versão serial observado entre a Tabela 5-4 para Tabela 5-4 é devido ao tempo de pós-processamento da agregação das séries. Este tempo está relacionado à totalização das séries e escrita dos arquivos, em todos os casos é maior que o próprio tempo de agregação paralela.

Tabela 5-3: Resultados da agregação das séries - Gevazp_sh - float

	Serial	Shared Memory	
		Tempo (s)	Speedup
	versao_2	sh_float	sh_float
Jan	32,247	1,491	21,6
Fev	32,422	1,753	18,5
Mar	40,41	1,722	23,5
Abr	39,572	1,896	20,9
Mai	49,685	1,906	26,1
Jun	65,542	2,044	32,1
Jul	165,86	4,502	36,8
Ago	84,217	2,212	**38,1**
Set	83,074	2,43	34,2
Out	64,144	1,925	33,3
Nov	53,743	1,971	27,3
Dez	36,361	1,573	23,1

Tabela 5-4: Tempo total e overall speedup - Gevazp_sh – float

	Serial	Shared Memory	
		Tempo (s)	Speedup
	versao_2	sh_float	sh_float
Jan	34,00	3,32	10,2
Fev	34,15	3,51	9,7
Mar	42,43	3,71	11,4
Abr	41,56	3,89	10,7
Mai	52,11	4,72	11,0
Jun	68,84	5,52	12,5
Jul	171,18	10,45	**16,4**
Ago	88,16	6,28	14,0
Set	86,58	6,07	14,3
Out	67,49	5,18	13,0
Nov	56,62	4,96	11,4
Dez	38,57	3,81	10,1

5.4 Resultados para um caso com 6000 aberturas

Para avaliar os ganhos de desempenho e a redução dos tempos em um caso com alta demanda computacional, modificou-se a configuração do um caso PMO de Setembro, com uma árvore de cenários composta por 30 aberturas no mês de Agosto e 200 aberturas no mês de Setembro, num total de 6000 aberturas Figura 5-15.

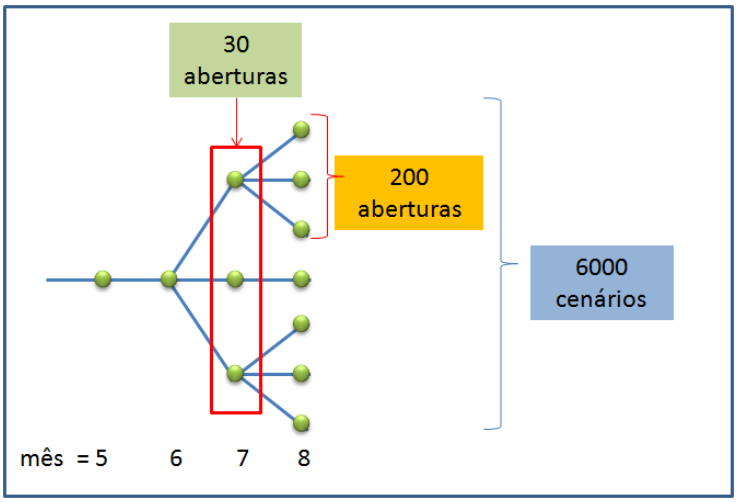


Figura 5-15: Árvore com 6000 cenários
30 aberturas no mês de Julho e 200 aberturas no mês de Agosto

Como observado na Tabela 5-5 foi obtido um *overall speedup* de **26,2 vezes**, reduzindo o tempo de **53,5 minutos para 2 minutos.**

Tabela 5-5: Tempo total e overall speedup - Gevazp_sh – float com 6000 aberturas

		Shared Memory	
	Serial	**Tempo (s)**	**Speedup**
	versao_2	sh_float	sh_float
Outubro	3210,7	122,4	26,2

5.5 Resultados para um caso com 6000 aberturas com a solução híbrida MPI e CUDA

Neste experimento, pode-se avaliar o desempenho da aplicação com até 6 processadores acessando simultaneamente a GPU, que é o número máximo de processadores disponíveis no computador. No mês 7 serão gerados 30 cenários no processo de agregação, ou seja, 1 processador será responsável por agrupar 1024 cenários em 30. Assim, neste primeiro mês não é possível se beneficiar da paralelização em MPI. Já no mês 8, cada um dos 30 cenários irá gerar 200 novos cenários, num total de 6000 cenários. Assim, pode-se dividir a tarefa dos 30 cenários pelo número de processadores utilizados pelo MPI, se beneficiando a geração de cenários em paralelo.

Este experimento tem o objetivo de avaliar dois pontos distintos:

- O ganho de desempenho do processo de agregação na GPU, com vários processadores utilizando os recursos da GPU simultaneamente;
- O ganho de desempenho total da versão híbrida.

A figura apresenta os tempos do processo de agregação e o tempo total de execução da versão híbrida executando de 1 a 6 processadores em paralelo, enquanto a figura apresenta os *speedups* obtidos.

O tempo de agregação foi reduzido de **78,9 s** com 1 processador para **16,8 s** com 6 processadores. Esse resultado demonstra que é possível obter um desempenho ainda melhor na utilização dos recursos da GPU utilizando-se *CUDA-Streams*, em vários processadores, reduzindo o tempo ocioso da GPU. Os *speedups* desta etapa tiveram uma característica linear, com uma eficiência média de 75%, mostrando um bom escalonamento.

O tempo total de execução foi reduzido de **122,4 s** com 1 processador para **74,9 s** com 3 processadores no melhor caso. Entretanto não se obteve ganhos no tempo total de execução com mais processadores. Esse resultado pode ser explicado por três fatores:

- A geração de séries para o primeiro mês de abertura (mês 7) é realizado por apenas 1 processador, contribuindo com uma parcela de tempo serial;
- O tempo de totalização das séries e a geração de arquivos também contribui com uma parcela de tempo serial;
- O tempo de *overhead* de comunicação aumenta à medida que o número de processadores aumenta.

Dessa forma, a partir de 3 processadores para o caso com 6000 aberturas, o tempo ganho gerando as séries em paralelo é compensado pelos tempos seriais e de *overhead* de comunicação, sendo esta a melhor configuração para este caso.

Em resumo, a solução híbrida foi capaz de reduzir o tempo de execução de **53,5 minutos** para **1 minuto e 14 segundos**, reduzindo o tempo total da aplicação em **42 vezes**, melhorando ainda mais o desempenho da solução em CUDA.

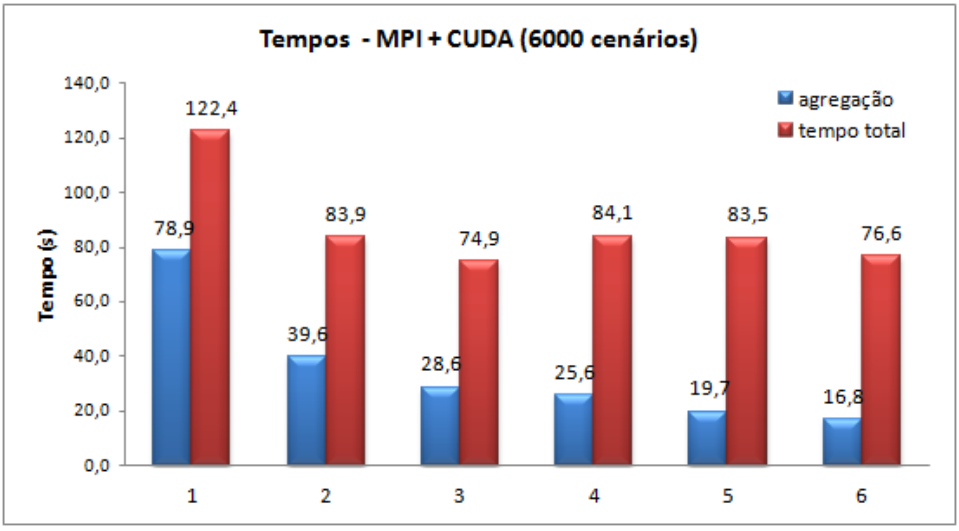


Figura 5-16: Tempos de agregação e total - árvore com 6000 cenários

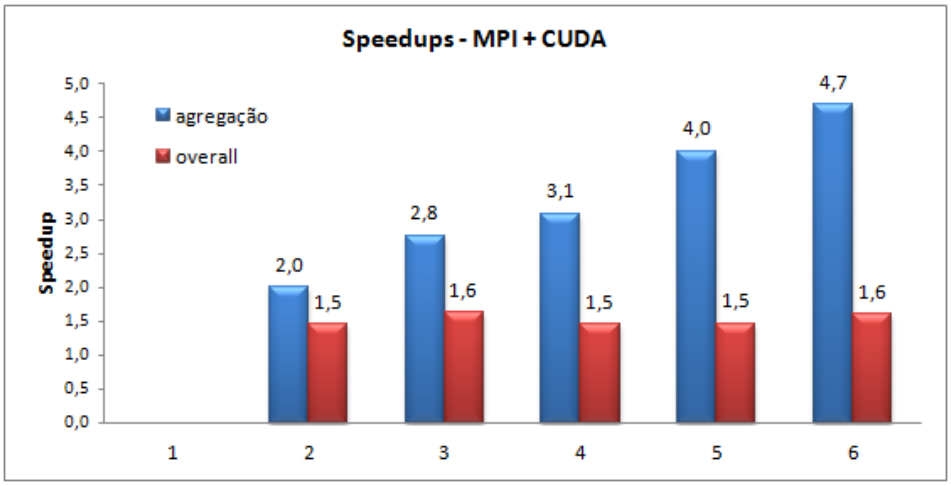


Figura 5-17: Speedups de agregação e total - Árvores com 6000 cenários

6 CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho apresentou uma metodologia otimizada de programação paralela em CUDA para redução de tempo computacional do modelo de geração de cenários sintéticos de energias e vazões – GEVAZP. A metodologia proposta atuou no processo de agrupamento de cenários baseado no algoritmo K-means. Na solução em CUDA, foi possível explorar a natureza paralela das operações entre vetores e matrizes para o cálculo das distâncias entre as vazões e os centróides dos agrupamentos. Na versão paralela, o cálculo das distâncias é feito em paralelo na GPU, e seu resultado é utilizado no algoritmo de agrupamento, na CPU.

Foram implementadas três versões para avaliar o potencial de processamento paralelo da GPU: Gevazp_gm que utiliza apenas a memória global, Gevazp_sh que utiliza memória compartilhada e Gevazp_CUBLAS que utiliza a biblioteca paralela de álgebra linear CUBLAS. Nessas três versões foram exploradas técnicas de otimização de código e de acesso às memórias global e compartilhada, com o objetivo de extrair o máximo desempenho da GPU. Nas três versões, foi avaliado o desempenho no uso de ponto flutuante de precisão simples e precisão dupla (4 e 8 *bytes* respectivamente). Os resultados numéricos foram idênticos utilizando-se nos dois casos com um ganho de desempenho maior para na utilização de precisão simples.

O uso das GPUs apresentou ganhos significativos no desempenho da aplicação, onde a versão utilizando memória compartilhada com precisão simples obteve em todos os casos o melhor desempenho computacional. A metodologia para avaliar o desempenho mediu os tempos de inicialização do algoritmo de agregação de cenários, os tempos de execução da rotina OPTRA, do algoritmo de agregação cenários e o tempo total de execução do programa.

Nas duas primeiras medidas foi possível atingir *speedups* de até 100 vezes. O processo de agregação apresentou *speedups* de até 38,1 vezes, e o tempo total de execução do programa foi reduzido em até 16,4 vezes, nos casos de PMO. No caso de 6000 cenários com maior demanda por processamento, foi possível obter uma redução no tempo total de execução de 26,2 vezes, reduzindo de **53,5 minutos para 2 minutos.**

Os resultados dos experimentos mostram que a versão utilizando memória compartilhada apresentou desempenhos até quatro vezes mais rápidos que as outras versões utilizando memória global e CUBLAS. Durante este trabalho foram estudadas técnicas de otimização no uso das GPUs, explorando conceitos como latência de acesso à memória,

acesso coalescente à memória global, compartilhamento de memória entre *threads*, conflito de banco de memória compartilhada, cooperação entre CUDA-*threads*, blocos de *threads*, níveis de ocupação e utilização de algoritmos de redução paralela. O conhecimento da arquitetura das placas GPU e o domínio desses conceitos é essencial para se desenvolver aplicações paralelas capazes de extrair o máximo desempenho desses dispositivos.

Além disso, foi apresentada também uma solução paralela utilizando *cluster* de computados em MPI que foi integrada com a solução em CUDA. Esta solução híbrida MPI-CUDA gera as séries em paralelo em cada processador de um *cluster*, e cada processador usa sua GPU para acelerar o processo de agregação de séries. Como só se dispunha de 1 GPU para testar os experimentos, utilizou-se *CUDA-Streams* para que vários processadores pudessem acessar os recursos da mesma GPU simultaneamente. Esta solução obteve um ganho ainda maior de desempenho. No caso de 6000 cenários foi possível obter uma redução no tempo de agregação de **78,9 para 16,8 segundos**, acelerando com 6 processadores **4,7** vezes em relação à solução em CUDA. A redução do tempo total da aplicação foi de **42 vezes,** reduzindo de **53,5 minutos** para **1 minuto e 14 segundos**.

A tecnologia CUDA aplicada ao modelo GEVAZP e integrada à bilioteca MPI apresentou resultados expressivos se mostrando uma tecnologia complementar na paralelização dos modelos de planejamento da geração utilizados no setor elétrico. Esses resultados motivam a utilização de GPU em outros modelos.

6.1 Trabalhos futuros

Frente aos bons resultados obtidos na paralelização do modelo GEVAZP em CUDA e na solução híbrida, faz-se necessária a continuação desta pesquisa em pontos que ainda podem ser explorados para obtenção de melhorias no desempenho computacional da versão paralela do modelo:

- No algoritmo de agregação de cenários, deve ser estudado o ponto em que a atualização dos centróides dos grupos deve ser feita para minimizar o número de iterações, mantendo ainda o algoritmo paralelizável.
- Executar o GEVAZP paralelo em GPUs com um maior número de SMs e de *CUDA-cores* para avaliar o desempenho da aplicação nesses dispositivos. Os

experimentos foram realizados utilizando-se uma GPU com 1 SM e 192 cores e 1 GB de memória RAM, com uma capacidade máxima de processamento de 336 Gigaflops. Atualmente a GPU K80 top de linha da NVIDIA, específica para o uso em HPC, possui um total de 4992 *CUDA-cores* e 24 GB de memória, com uma capacidade de processamento de até 8,73 Teraflops;

- Utilizar um cluster de GPUs, em casos com maior demanda computacional, para avaliar o desempenho da aplicação nesta configuração e buscar melhorias na solução em MPI.

E por fim, recomenda-se o estudo da aplicação da tecnologia CUDA em outros modelos de planejamento da operação.

BIBLIOGRAFIA

[1] **Moore, Gorgon E., "***Cramming more components into integrated circuits".* 8, April 9, 1965, Electronics Magazine, Vol. 38, p. 4.

[2] **Roberto J. Pinto, Carmen L.T. Borges, Maria E. P. Maceira.** "An Efficient Paralell Algorithm for Large Scale Hydrotermal System Operation Planning". *IEEE Transactions on Power Systems,*. 2011, Vols. 28, No 4, pag. 4888 - 4896.

[3] **M.E.P. Maceira, V.S. Duarte, D.D.J. Penna, L.A.M. Moraes, A.C.G. Melo**, "Ten years of application of stochastic dual dynamicProgramming in official and agent studies in Brazil –Description of the NEWAVE program", *16th Power Systems Computation Conference - PSCC*, Glasgow, SCO, July 2008.

[4] **Quinn, Michael J.** "*Parallel Programming in C with MPI and OpenMP".* Oregon, Oregon State University : s.n., 2004.

[5] **Nvidia Corporation.** "CUDA Toolkit Documentation - Programming Guide". [Online] 2016. http://docs.nvidia.com/cuda/cuda-c-programming-guide.

[6] **M. E. Maceira, Bezerra and C. V.** "*Stochastic Streamflow model for Hydroelectric Systems*". Vancouver, Canada : s.n., 1997. 5th Int. Conf. on Probabilistic Methods Applied to Power Systems -PMAPS.

[7] **NVIDIA Coorporation.** "CUDA - CUBLAS Library 2.0". [Online] https://developer.nvidia.com/cublas.

[8] **Barney, Blaise.** "Introduction to Parallel Computing". [Online] Lawrence Livermore National Laboratory. https://computing.llnl.gov/tutorials/parallel_comp/#Terminology.

[9] **Flynn, M. J.** "*Some Computer Organizations and Their Effectiveness".* IEEE Transactions on Computers, Semptember de 1972, pp. C–21 (9): 948–960.

[10] **Nvidia.** "What is GPU computing?" [Online] NVidia Coorporation, 2015. http://www.nvidia.com/object/what-is-gpu-computing.html.

[11] **Assam, Amr H.** "GPU computing for beginners" (HPC and Webinar Series). [Online] Swimburn University of Technology, 2014. https://www.youtube.com/watch?v=BPjgwRftoNo.

[12] **NVIDIA.** "NVIDIA's Next Generation, CUDA Compute Architecture: Kepler GK110" [Online] NVIDIA, 2012. https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[13] **John Cheng, Max Grossman, Ty McKercher.** "Professional CUDA C Programming". s.l. : John Wiley & Sons, 2014.

[14] **NVidia Corporation.** "NVidia CUDA Toolkit". [Online] NVidia Corporation, 2016. https://developer.nvidia.com/cuda-toolkit.

[15] **Conrad, Danilo Fukuda.** "*Análise da Hierarquia de Memória em GPGPUs".* s.l. : UFRJGS, 2010.

[16] **NVidia Coorporation.** "Archieved Ocupancy". [Online] 2015. http://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm.

[17] **L. A. M. Fortunato, T. A. A. Neto, J. C. R. Albuquerque, M. V. F. Pereira.** "*Introdução ao planejamento da expansão e operação de sistemas de produção de energia elétrica".* s.l. : Niterói, Universidade Federal Fluminense, EDUFF, 1990.

[18] ***E. L. Silva.*** *"*Formação de preços em mercados de energia elétrica**".** s.l. : Ed. Sagra Luzatto, 2001.

[19] **L.A.Terry, M.V.F.Ferreira, T.A.Araripe Neto, L.F.C.A.Silva, P.R.H.Sales.** "Coordinating the Energy Generation of the Brazilian National Hydrothermal Electrical Generating System". January-February 1986, Vol. 16.

[20] **ONS.** "Estrutura da Capacidade Instalada no SIN – MW". [Online] ONS, 2015. www.ons.org.br.

[21] **ONS** – "Operador Nacional do Sistema Elétrico". [Online] 2015. http://www.ons.org.br/.

[22] **Silva, E.L. da., Finardi, E.C.,** "Curso de Planejamento da Operação de Sistemas Hidrotérmicos", Projeto CEPEL / ASMAE / UFSC, Rio de Janeiro, Novembro 1999.

[23] **PENNA, D. D. J., MACEIRA, M. E. P. e DAMÁZIO, J. M.** "Streamflow Scenario Tree: Selective Sampling in the Long-Term Generation Planning". In: Proceedings of 10th Symposium on Stochastic Hydraulics and 5th International Conference on Water Resources and Environment Research, Quebec City, Canada : s.n., Jul 2010.

[24] **MACEIRA, M. E. P. et al.** "*Chain of optimization models for Setting de Energy Dispatch and Spot Price in Brazilian System".* s.l. : 14h PSCC, 2002.

[25] *Application of CVaR risk aversion approach in the expansion and operation planning and for setting the spot price in the Brazilian hydrothermal interconnected system.* Nov. 2 : s.n., Nov. 2015, International Journal of Electrical Power and Energy Systems, Vol. 72, pp. 126-125.

[26] *"Desenvolvimento, implementação e testes de validação das metodologias para internalização de mecanismos de aversão a risco nos programas computacionais para estudos energéticos e formação de preço", Relatório Técnico da CPAMP, Julho 2003,*

[27] **L.A. Terry, M.E.P. Maceira, C.V. Mercio, V.S. Duarte.** "*Equivalent reservoir model for hydraulic coupled systems".* Rio de Janeiro, Brasil : s.n., May, 2004.

[28] **N. V. Arvantidis, J. Rosing,.** "*Composite representation of multireservoir hydroelectric power system".* 2, Feb. 1970, IEEE Transactions on Power Apparatus and Systems, Vol. 89, pp. 319-326.

[29] **MACEIRA, M.E.P., CUNHA, S.H.F.** "*Simulating the Energy Generation of Interconnect Hydro-Thermal Systems – SUISHI Model".* Balneário de Camburiú, Brasil : Proceedings of XIII Seminário Nacional de Produção e Transmissão de Energia Elétrica, 1995.

[30] **R. J. Pinto, A.L.G.P. Sabóia, R.N. Cabral, F.S. Costa, A.L.Diniz e M. E. P. Maceira,** "Metodologia para aplicação de processamento paralelo no planejamento de curto-prazo (modelo DECOMP)", *XX SNPTEE- Seminário Nacional de Produção e Transmissão de Energia Elétrica*, Recife, Novembro 2009.

[31] **R. J. Pinto, A.L.G.P. Sabóia, R.N. Cabral, F.S. Costa, A.L.Diniz e M. E. P. Maceira.** "*Metodologia para aplicação de processamento paralelo no planejamento de curto-prazo (modelo DECOMP)".* Recife : s.n., Novembro, 2009. XX SNPTEE- Seminário Nacional de Produção e Transmissão de Energia .

[32] **JARDIM, D.L.D.D, MACEIRA, M.E.P., FALCÃO, D.M.,** 2001, "Stochastic Streamflow model for Hydroelectric Systems Using Clustering Techniques", 2001 IEEE Porto Power Tech Conference, Porto, Portugal, Sep.

[33] **MACEIRA, M.E.P.** "*Operação Ótima de Reservatórios com Previsão de Afluências".* Rio de Janeiro,Brasil. : Dissertação de Mestrado, COPPE/UFRJ, 1989.
34. *A K-Means Clustering Algorithm.* **Wong, J. A. Hartigan and M. A.** 1, 1979, Journal of the Royal Statistical Society, Vol. 28, pp. 100-108.

[35] **Johnson, R. e Wichern.** *Applied Multivariate Statistical Analysis.* s.l. : Prentice Hall, 1998. item 1.5.

[36] **PENNA, D. D. J.; MACEIRA, M. E. P.; DAMÁZIO, J. M.,** 2005, "Geração de Cenários Sintéticos de Energia e Vazão para o Planejamento da Operação Energética", In: Proceedings of XVI Simpósio Brasileiro de Recursos Hídricos, Nov.

[37] **BOX, G.E.P., JENKINS, G.M.** "*Time Series Analysis - Forecasting and Control".* San Francisco : Holden - Day, 1976.

[38] **J.M. Latorre, S. Cerisola, A. Ramos.** "*Clustering algorithms for scenario tree generation: Application to natural hydro inflows".* European Journal of Operational Research, v.181, n.3, pp. 1339-1353, 2007.

[39] **Haris, Mark.** "Optimizing Paralell Reduction in CUDA". [Online] NVidia Coorporation, 2007. http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf.

[40] "How to Access Global Memory Efficiently in CUDA C/C++ Kernels". [Online] 07 de Janeiro de 2013. https://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/.

[41] **Netlib.** “BLAS – basic linear algebra subprograms”. [Online] Netlib. http://www.netlib.org/blas.

[42] **Spampinato, Danielle Giuseppe.** “*Linear Optimization with CUDA”.* Norway : s.n., 2009, January.

[43] **NVidia Corporation.** “Cuda C Best Practices Guide”. [Online] 2016. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide.

[44] **Wittek, Peter.** “Cublas Multiplication Matrix with C-Style Arrays”. [Online] 2013. http://peterwittek.com/cublas-matrix-c-style.html.

[45] **Luitjens, Justin.** “Faster Paralell Reductions on Kepler”. [Online] NVidia Coorporation, 2014. https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler.

[46] **Silva, E. L.** “*Formação de preços em mercados de energia elétrica”.* s.l. : Ed. Sagra Luzatto, 2001.

[47] Centro de Pesquisas de Energia Elétrica. [Online] www.cepel.br.

[48] **PROCEL.** “*Avaliação do mercado de eficiência energética no Brasil”.* Rio de Janeiro : Eletrobras Procel, 2007.

ANEXO A

ALGORITMO DE AGRUPAMENTO DE HARTIGAN & WONG

Considere um conjunto de dados contendo *M* observações de dimensão *N*. O objetivo é classificá-las em *K* grupos.

Seja *NC(L)* o número de elementos do grupo *L* e *D(I, L)* a distância entre a observação *I* e a média do grupo *L*.

Forneça um conjunto de *K* vetores *N*-dimensionais como valores iniciais para as *K*-médias.

Passo 1: Para cada ponto *I (I = 1, ..., M)*, encontre a sua média mais próxima e sua segunda média mais próxima, *IC1(I)* e *IC2(I)*, respectivamente. Atribua o ponto *I* ao grupo *IC1(I)*.

Passo 2: Atualize as médias dos grupos para serem as médias dos pontos contidos dentro deles.

Passo 3: Inicialmente, todos os grupos pertencem ao conjunto ativo.

Passo 4: (*Optimal transfer stage*): Considere cada ponto *I (I = 1, ... , M)*. Se o grupo *L (L = 1, ... , K)* foi atualizado no passo 6, então ele pertence ao conjunto ativo. Caso contrário, em cada passo, ele não está no conjunto ativo se não foi atualizado nos últimos *M* passos do passo 4. Seja *L1* o grupo do ponto *I*. Se *L1* está no conjunto ativo, vá para o passo 4ª. Caso contrário, vá para o passo 4b.

Passo 4a: Calcule o mínimo da quantidade $R1 = \frac{NC(L1)D(I,L)^2}{NC(L1)-1}$ e $R2 = \frac{NC(L)D(I,L)^2}{NC(L)+1}$, sobre todos os grupos *L (L ≠ L1, L = 1, ... , K)*. Seja *L2* o grupo com menor *R2*. Se *R2* é maior ou igual a *R1*, a realocação não será necessária e *L2* é o novo *IC2(I)*. Note que *R1* é relembrado e permanecerá o mesmo para o ponto *I* até que *L1* seja atualizado.

Caso contrário, o ponto *I* é alocado ao grupo *L2* e *L1* é o novo *IC2(I)*. As médias dos grupos são atualizadas para serem as médias dos pontos atribuídos a eles se a realocação ocorreu. Os dois pontos envolvidos na transferência do ponto *I* neste passo estão agora no conjunto ativo.

Passo 4b: Este passo é idêntico ao 4ª, exceto que o mínimo de *R2* é calculado somente sobre os grupos no conjunto ativo.

Passo 5: Pare se o conjunto ativo estiver vazio. Caso contrário vá para o passo 6.

Passo 6: (*Quick transfer stage)*: Considere cada ponto *I (I = 1, ... , M).* Fala *L1 = IC1(I)* e *L2 = IC2(I)*. Não é necessário checar o ponto *I* se ambos os grupos *L1* e *L2* não mudaram nos últimos *M* passos. Calcule $R1 = \frac{NC(L1)D(I,L)^2}{NC(L1)-1}$ e $R2 = \frac{NC(L)D(I,L)^2}{NC(L)+1}$. Se *R1* é menor que *R2*, o ponto *I* permanece no grupo *L1*. Caso contrário, troque *IC1(I)* com *IC2(I)* e atualize as médias dos grupos *L1* e *L2*. Os dois grupos são também notados por seu envolvimento numa transferência neste passo.

Passo 7: Se nos últimos *M* passos nenhuma transferência foi realizada, vá para o passo 4. Caso contrário, vá para o passo 6.