Orlando Saraiva do Nascimento Júnior

Técnicas de computação paralela aplicadas ao Método das Características em Sistemas Hidráulicos

Parallel Computing applied to Method of Characteristics in Hydraulic Systems

Limeira, SP

2013

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE TECNOLOGIA

Orlando Saraiva do Nascimento Júnior

Técnicas de computação paralela aplicadas ao Método das Características em Sistemas Hidráulicos

Orientador: Prof. Dr. Vitor Rafael Coluci

Co-orientadora: Profa. Dra. Lubienska Cristina Lucas Jaquiê Ribeiro

Parallel Computing applied to Method of Characteristics in Hydraulic Systems

Dissertação de Mestrado apresentada ao Programa de Pós Graduação em Tecnologia da Faculdade de Tecnologia da Universidade Estadual de Campinas para obtenção do título de Mestre em Tecnologia
Área de concentração: Tecnologia e Inovação.

Master degree dissertation presented to the graduate studies program of Faculty of Technology of the University of Campinas to obtain the Master grade in Technology.
Concentration Area: Technology and Innovation.

Limeira, SP

2013

FICHA CATALOGRÁFICA ELABORADA POR VANESSA EVELYN COSTA CRB-8/8295
BIBLIOTECA UNIFICADA FT/CTL
UNICAMP

Nascimento Júnior, Orlando Saraiva, 1981-
N17t Técnicas de computação paralela aplicadas ao método das características em sistemas hidráulicos / Orlando Saraiva do Nascimento Júnior. -
Limeira, SP : [s.n.], 2013.

Orientador: Vitor Rafael Coluci.
Coorientador: Lubienska Cristina Lucas Jaquiê Ribeiro.
Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Tecnologia.

1. Programação paralela (Computação). 2. Computadores paralelos. 3. Simulação (Computadores) – dinâmica dos fluídos. 4. Simulação (Computadores) – modelos matemáticos. I. Coluci, Vitor Rafael. II. Ribeiro, Lubienska Cristina Lucas Jaquiê. III. Universidade Estadual de Campinas. Faculdade de Tecnologia. IV. Título.

Informações para Biblioteca Digital

Título em inglês: Parallel computing applied to method of characteristics in hydraulic systems.

Palavras-chave em inglês (Keywords):

1- Parallel programming (Computing)
2- Parallel computers
3- Simulation (computers) – fluid dynamics
4- Simulation (computers) – mathematical models

Área de concentração: Tecnologia e Inovação

Titulação: Mestre em Tecnologia

Banca examinadora: Vitor Rafael Coluci, Edevar Luvizotto Junior, Fábio Roberto Chavarette.

Data da Defesa: 22-02-2013

Programa de Pós-Graduação em Tecnologia

DISSERTAÇÃO DE MESTRADO EM TECNOLOGIA

ÁREA DE CONCENTRAÇÃO: TECNOLOGIA E INOVAÇÃO

Técnicas de computação paralela aplicadas ao Método das Características em Sistemas Hidráulicos

Orlando Saraiva do Nascimento Júnior

A Banca Examinadora composta pelos membros abaixo aprovou esta Dissertação:

Vitor Rafael Coluci

Prof. Dr. Vitor Rafael Coluci, Presidente
FT/UNICAMP

Prof. Dr. Edevar Luvizotto Junior
FEC/UNICAMP

Prof. Dr. Fábio Roberto Chavarette
FEIS/UNESP

Agradecimentos

À Deus, primeiramente, por me abençoar em todos os meus passos.

Agradeço a minha esposa, Ana, pela enorme paciência e compreensão, compartilhando os bons e maus momentos da realização deste trabalho. Foram muitos os sacrifícios e madrugadas. Escrevendo, lendo, codificando. Sem sua compreensão e seu incondicional apoio, nada disso seria possível.

Agradeço ao meu filho, Lucas. Sua vinda ao mundo durante a execução deste trabalho fortaleceu ainda mais minha vontade de lutar e progredir.

Agradeço aos meus familiares e amigos. Pais, irmão, cunhadas, tios e tias, primos e primas, sobrinhos e sobrinhas, afilhada e afilhado. A todos, pelos pequenos e grandes gestos de apoio, torcida e carinho. Uma palavra, um sorriso, uma mensagem nas redes sociais. Foram muitas as formas de apoio.

Ao meu orientador, Prof. Dr. Vitor Rafael Coluci, por seu exemplo e dedicação como professor e orientador. Agradeço por acreditar no meu trabalho, pela paciência em me ensinar os caminhos da Ciência e a importância da escrita científica.

À minha co-orientadora, Profa. Dra. Lubienska Cristina Lucas Jaquiê Ribeiro, pela paciência em entender minhas deficiências em uma nova área e por aceitar o desafio do trabalho interdisciplinar.

Aos funcionários e professores da Faculdade de Tecnologia (FT/Unicamp), por todo apoio e suporte via *facebook* em todas as horas.

Aos funcionários da Unesp Rio Claro presentes nas etapas do meu trabalho. À todos os colegas da FHO Uniararas, em especial ao amigo e prof. Fábio do núcleo de Engenharia, grande incentivador desta caminhada.

.

Para Ana e para Lucas, com amor.

.

Resumo

Uma instalação hidráulica é um conjunto de dispositivos hidromecânicos e tubos com a função de transportar um fluido. O controle do escoamento deste fluido ocorre por meio de manobras nos dispositivos hidromecânicos. Uma investigação sobre o impacto das manobras destes dispositivos em uma instalação hidráulica pode evitar danos físicos ao sistema (como rompimento de tubos, por exemplo).

Uma das formas de se investigar o efeito destas manobras é por meio da simulação. A simulação permite estudar um sistema hidráulico, que após uma manobra hidráulica sai de uma situação contínua (regime permanente inicial), entra em um estado transitório (regime transiente) para posteriormente entrar em uma nova situação contínua (regime permanente final). No regime de transiente hidráulico são formadas ondas de sobrepressão e subpressão internas na tubulação e que podem levar a danos.

Um dos métodos mais aceitos para simulações de transiente hidráulico é o método das características, que permite transformar as equações diferenciais parciais que descrevem o fenômeno em um conjunto de equações diferenciais ordinárias. Dependendo do tamanho do sistema hidráulico (número e comprimento de tubos, número de dispositivos eletromecânicos, etc), o custo computacional pode ser elevado para se obter as informações sobre o comportamento do transiente.

Neste trabalho aplicamos técnicas de computação paralela em placas de vídeos para processamento de propósito geral (GPU) e em multi-núcleos (OpenMP) para acelerar os cálculos do transiente hidráulico. Utilizamos um sistema hidráulico composto por um reservatório, uma válvula e um tubo e determinamos o ganho de desempenho em função do tamanho do tubo do sistema.

A técnica OpenMP forneceu ganhos computacionais de até $3.3\times$ enquanto a técnica envolvendo GPUs forneceu ganhos de $17\times$. Dessa forma, placas gráficas se mostraram muito interessantes para acelerar simulações de transientes hidráulicos com o método das características.

Palavras-chave: Transiente Hidráulico, Computação Paralela, Método das Características, Arquitetura CUDA.

.

Abstract

A hydraulic system is a set of hydromechanical devices and tubes designed to transport fluids through controlled operations. Investigating the impact of these operations on hydraulic systems can avoid physical damage to its parts (such as breakage of pipes, for example).

One way to investigate these impacts is through computational simulations. The simulations allow to study a hydraulic system during initial and final steady states (after some device operation, for instance), and the transient state between them. During the hydraulic transient state, high and low pressure waves are formed in the tubes and are the main cause of tube damages.

One of the most accepted method for transient hydraulic simulations is the method of characteristics, which allows to transform the partial differential equations that describe the phenomenon in a set of ordinary differential equations. Depending on the size of the hydraulic system (number and length of tubes, number of electromechanical devices, etc), the computational cost to obtain information about the behavior of the transient can be large.

In this work, we apply techniques of parallel computing involving video cards for general purpose processing (GPU) and multi-cores (OpenMP) to accelerate hydraulic transient calculations. We simulated a hydraulic system consisting of a reservoir, a valve and a pipe to determine the performance speedup as a function of the size of the pipe.

The OpenMP technique provided computational speedup up to $3.3\times$ whereas the GPU technique provided speedup of $17\times$. Therefore, our results indicated that GPUs are very interesting to accelerate hydraulic transients simulations using the method of characteristics.

Keywords: Hydraulic transients, Parallel Computing, Method of characteristics, CUDA architecture.

Sumário

Lista de Figuras **xvii**

1 Introdução **1**

2 Método das Características **5**

2.1 Sistema Hidráulico Simulado 5

2.2 Transiente Hidráulico 6

3 Computação Paralela **13**

3.1 Computação Paralela 13

3.2 Arquitetura GPU 16

3.3 Métricas para análise de desempenho 21

4 Estratégias de paralelismo **23**

4.1 Desenvolvimento Serial 23

4.2 Desenvolvimento Paralelo 29

5 Simulações e Resultados **41**

5.1 Cenários dos testes e validação 41

5.2 Análises 44

6 Conclusões **51**

7 Perspectivas futuras **53**

Referências bibliográficas **54**

A Código Fonte **59**

A.1 versão C serial 59

A.2 versão CUDA-C 65

A.3 versão OpenMP 73

A.4 Para comparar resultados obtidos. 80

Trabalhos Publicados Pelo Autor **85**

Lista de Figuras

2.1 Sistema Reservatório − Tubo − Válvula. 5

2.2 Reservatório e tubo observados na cidade de Mogi Mirim/SP. 6

2.3 Ilustração do histórico da Pressão nos segmentos X e Y ao longo do tempo. Ao lado esquerdo, a visualização de ondas de sobrepressão e subpressão e os dois segmentos observados. Ao lado direito, dois gráficos ilustram o histórico da pressão nos dois segmentos - X e Y - em destaque. Por ser uma ilustração, desconsidera a vaporização que ocorreria nos valores apresentados. 7

2.4 Representação do espaço discretizado $x \times t$. Em azul, as características nos pontos anteriores (C+) e posteriores (C-) no tempo anterior (tempo 0) são utilizadas para encontrar o valor no ponto vermelho no tempo posterior (tempo 1). 11

3.1 Visão geral da arquitetura de Von Neumann (adaptado de [18]) 14

3.2 Visão geral de um sistema de memória compartilhada (adaptado de [18]) 15

3.3 Visão geral de um sistema de memória distribuída (adaptado de [18]) 15

3.4 Visão geral da arquitetura GPU. 16

3.5 Exemplo de *thread*, bloco, *grid* e *kernel*. Adaptado de [27]. 17

3.6 Hierarquia de memórias em CUDA (Adaptado de [25]). 18

3.7 Exemplo de função em C com uso de CUDA-C. Neste exemplo, aloca-se três vetores de 2000 posições em *host* e três vetores em *device* OS vetores A e B são copiados do *host* para *device*, a soma é calculadas via *kernel* com uso de 20 blocos e 100 *threads* em cada bloco. O resultado da soma, alocado no vetor C, copiado do *device* para *host*. 19

3.8 Exemplo de código C com uso de OpenMP. Neste exemplo, aloca-se três vetores de 2000 posições. O vetor A, B, C e a variável chunk são compartilhadas por todas as *threads*. 21

4.1 Exemplo do arquivo de entrada 24

4.2 Pseudo-código da implementação serial. 25

4.3 Representação das características necessárias dentro da malha computacional. 26

4.4 Implementação Serial conforme computação ocorre: Em (a) ocorre o cálculo da pressão e vazão no segmento 3 no tempo 1 com uso da pressão e vazão do tempo 0 no segmento 2 (C+) e no segmento 4 (C-). Os segmentos 4, 5, 6 no tempo 1 são calculados respectivamente em (b), (c) e (d). 28

4.5 Implementação paralela com a API OpenMP, utilizando-se duas threads: Em (a), duas threads calculam os segmentos 1 e 2 no tempo 1. A primeira thread, no segmento 1 utiliza C+ do segmento 0 no tempo 0 e C- do segmento 2 no tempo anterior. A segunda thread, no segmento 2 utiliza C+ do segmento 1 no tempo 0 e C- do segmento 3 no tempo anterior. Em (b), as duas threads calculam os segmentos 3 e 4 no tempo 1. Em (c), duas threads calculam os segmentos 5 e 6. Em (d), os segmentos 7 e 8 são calculados paralelamente. Os segmentos calculados são marcados em (b), (c) e (d) com a cor rosa. 31

4.6 Primeira implementação com uso da arquitetura CUDA. Em (a), as threads calculam os segmentos impares no tempo 1 (esquerda) e posteriormente os segmentos pares tempo 1 (direita). Em (b) as threads calculam os segmentos impares no tempo 2 (esquerda) e posteriormente os segmentos pares tempo 2. 33

4.7 Segunda implementação CUDA: com ponto de sincronização. Em (a), em destaque, o bloco e na direita os segmentos que serão trabalhados até a próxima sincronização. Em (b),no tempo t0, duas threads processando os segmentos 1 e 2 no tempo 1. Em t1, duas threads calculam os segmentos 3 e 4 no tempo 1, e em t2, o segmento 1 no tempo 1 é calculado. Em (c), o bloco destacado em (a), com destaque para os segmentos calculados.Os segmentos calculados são marcados em (b) e (c) com a cor rosa. 38

4.8 Segunda implementação CUDA: visão global dos blocos. Em (a), a seleção do bloco 1. Em (b) os segmentos calculados no bloco 1. Em (c), a seleção do bloco 2. Em (d), os segmentos calculados e a seleção do bloco 3. Os segmentos calculados são marcados em (b), (c) e (d) com a cor rosa. 39

5.1 Fluxograma com os passos realizados nos testes. 42

5.2 Comparativo dos resultados serial e paralelo via OpenMP em um determinado segmento. Em destaque o módulo da subtração dos resultados obtidos serialmente com os resultados obtidos de forma paralela. No canto inferior direito, uma visualização dos resultados obtidos de forma serial. 42

5.3 Comparativo dos resultados serial e paralelo via GPU em um determinado segmento. Em destaque o módulo da subtração dos resultados obtidos serialmente com os resultados obtidos de forma paralela. No canto inferior direito, uma visualização dos resultados obtidos de forma serial. 43

5.4 Tempo Serial: Total x Parcial 44

5.5 Tempo Serial: Total x Parcial (início) 44

5.6 Médias de tempo por segmentos com uso de OpenMP. O desvio padrão é apresentado nas médias com 4 e 128 threads apenas. 45

5.7 Médias de tempo por segmentos com uso da arquitetura CUDA. 46

5.8 Médias de tempo por segmentos: Serial x OpenMP x GPU 47

5.9 Comparação da eficiência computacional OpenMP com 128 *threads* x GPU com 16 *threads* por bloco 48

5.10 *Speedup*: OpenMP com 128 *threads* x GPU com 16 *threads* por bloco 49

5.11 Exemplo de mapeamento para sistemas maiores e o vetor representando o sistema como um todo. Em vermelho, os elementos representam o tubo 1, em amarelo o tubo 2 e em rosa o tubo 3. Os elementos em azul representam os nós de montante e jusante. 49

Capítulo 1

Introdução

Uma instalação hidráulica é um conjunto de dispositivos hidromecânicos e condutos destinados ao transporte de um fluido. Os dispositivos hidromecânicos tem a função de controlar o escoamento e transformar energia mecânica em energia hidráulica ou vice/versa [1]. Como exemplo desses dispositivos hidromecânicos podemos citar válvulas, comportas, turbinas, reservatórios, bombas, dentre outros.

Para o controle do escoamento de fluido, algumas manobras nos dispositivos hidromecânicos podem ser realizadas. Por exemplo, uma válvula presente em uma tubulação pode ser fechada para interromper o fluxo ou ser parcialmente fechada para controlar a vazão de um fluido. No entanto, manobras nos dispositivos hidromecânicos, podem levar a grandes variações de pressão e vazão no escoamento do fluido em um curto intervalo de tempo. Essas variações de vazão e pressão são conhecidas como transientes hidráulicos e podem levar a consequencias indesejáveis nas instalações hidráulicas como fluxos reversos, pressões elevadas ou negativas. O desconhecimento dos efeitos deste fenômeno pode ocasionar superdimensionamento ou subdimensionamento em projetos de sistemas de tubulações com espessuras de parede desnecessariamente elevadas ou perigosamente reduzidas.

Uma maneira de se investigar os efeitos dos transientes hidráulicos é por meio de simulação computacional onde as equações diferenciais que governam os fenômenos envoluvidos são resolvidas numericamente. No caso dos transientes hidráulicos, as equações que descrevem o comportamento da vazão e da pressão em diferentes regiões do sistema e em diferentes instantes de tempo são as equações diferenciais parciais do movimento e da continuidade. Diferentes métodos podem ser utilizados

para resolver essas equações como o método das diferenças finitas, método dos elementos finitos, método das características, dentre outros [2], [3]. No Brasil, a norma NBR 12215 [4] recomenda o método das características para a investigação numérica de fenômenos envolvendo transientes hidráulicos.

A ideia central do método das características é transformar as equações diferenciais parciais em equações diferenciais ordinárias que podem ser mais facilmente tratadas numericamente. Quando o sistema hidráulico investigado passa pelo processo de discretização, ou seja, as variáveis dependentes (pressão e vazão) são calculadas e armazenadas em pontos de uma malha de variáveis independentes (tempo e espaço), as equações obtidas pelo método das características são transformadas em equações de diferenças finitas. Dependendo do tamanho do sistema hidráulico (comprimento de tubulações, tempo desejado de simulação, etc), a resolução das equações pode ser computacionalmente custosa. Isso porque, em cada passo do tempo de simulação necessita-se obter os valores de pressão e vazão em cada ponto da malha. Para sistemas hidráulicos relativamente pequenos, simulações esporádicas, o custo computacional não se torna um problema. No entanto, quando se desejar simular sistemas grandes ou realizar muitas simulações do mesmo sistema em diferentes condições, o custo computacional pode se tornar um aspecto relevante a se considerar.

O custo computacional pode ser reduzido basicamente de duas maneiras. A mais simples se baseia na melhoria dos desempenhos das unidades centrais de processamento (CPUs), que independe da implementação do código computacional. Embora seja mais simples, a indústria de *hardware* tem mostrado pouco interesse neste forma de melhora. Em 2004, por exemplo, a Intel abandonou o desenvolvimento de dois novos processadores [5] por problemas como o aumento da temperatura no processador e aumento do consumo energético. Uma segunda maneira de reduzir o custo computacional é por meio de técnicas de computação paralela, que permitam modificar o código computacional afim de aproveitar diferentes tipos de arquiteturas. Uma dessas técnicas de computação paralela desenvolvida recentemente é a técnica baseada em unidades de processamento gráfico (GPUs-*Graphic Processing Units*)[6]. Essas unidades de processamento se apresentam como um novo *hardware*, que pode ser utilizado em cálculos numéricos.

O modelo GPU prevê usar CPU e GPU em um modelo de computação de processamento heterogêneo [6]. A parte sequencial da aplicação funciona na CPU e a parte computacionalmente intensa é

acelerada pela GPU. Enquanto CPUs modernas possuem no máximo 8 núcleos, uma GPU é formada por centenas, o que a torna especializada em tarefas de computação intensiva e algoritmos altamente paralelizáveis. Atualmente, existem três fornecedores de *hardware* no mercado de GPU's para propósito geral: NVIDIA, AMD e INTEL [7]. A empresa NVIDIA, além da pioneira e possuir a maior fatia deste mercado, criou a arquitetura CUDA: um conjunto de ferramentas de *softwares* para melhor integração ao seu *hardware*. Nesta arquitetura, dentre outros elementos de *software*, se prevê o uso da linguagem de programação CUDA-C, uma versão estendida da linguagem C-ANSI [8] e da linguagem de programação OpenCL, padrão aberto para programação paralela em sistemas heterogêneos [9].

O uso de GPU tem apresentado ganhos de desempenho expressivos em trabalhos envolvendo sistemas hidráulicos. Por exemplo, Wu e Eftekharian [10] utilizaram redes neurais artificiais paralelas para extração de conhecimento hidráulico em domínio de grandes sistemas de distribuição da água com expressivos ganhos e Crous, van Zyl e Nel [11] destacaram o potencial para ganhos significativos na velocidade de resolver equações hidráulicas de distribuição de água, mesmo com trabalhos em andamento.

O principal objetivo deste trabalho foi aumentar o desempenho computacional visando diminuir o tempo dos cálculos do transiente hidráulico pelo método das característcas. Para isto, foi necessário entender as restrições computacionais envolvidas no método, o que permitiu desenvolver estratégias para o uso otimizado das técnicas de computação paralela estudadas.

A abordagem hidráulica do transiente hidráulico é o objeto de outros trabalhos [12] [13] [14]. O presente trabalho realiza uma abordagem computacional sobre este problema. Este foco computacional utiliza a arquitetura CUDA [8] e a *API* de paralelismo OpenMP [15].

No capítulo 2, apresentamos uma breve explicação sobre a modelagem de um sistema hidráulico com resolução pelo método das características. No capítulo 3 apresentamos uma visão global da computação paralela, passando pelas principais classificações. Também é apresentado uma visão sobre GPGPU e os principais aspectos da arquitetura CUDA. Neste capítulo, será apresentado a *API* OpenMP, justificando sua escolha como parâmetro de comparação. No capítulo 4 apresentamos o processo de desenvolvimento de uma aplicação paralela e as estratégias de paralelismo escolhidas para o problema proposto. No capítulo 5 apresentamos o ambiente computacional GPU, os resulta-

dos em ganhos computacionais para cada estratégia proposta, discutindo a eficácia de cada estratégia. No capítulo 6 apresentamos as conclusões e por fim, no capítulo 7 apresentamos perspectivas futuras para o trabalho apresentado. Complementando, o apêndice A apresenta os códigos-fonte das implementações serial, CUDA-C e OpenMP, discutidas no capítulo 4.

Capítulo 2

Método das Características

Este capítulo se dedica em explicar o sistema hidráulico simulado e como se dá a aplicação do método das características na simulação do fenômeno hidráulico conhecido como Transiente Hidráulico.

2.1 Sistema Hidráulico Simulado

O sistema hidráulico escolhido para as simulações neste trabalho é um sistema mais simples possível, contendo um reservatório, um tubo e uma válvula. Uma das extremidades do tubo está conectado no reservatório (nó de montante) e a outra extreminadade na válvula (nó de juzante)(figura 2.1).

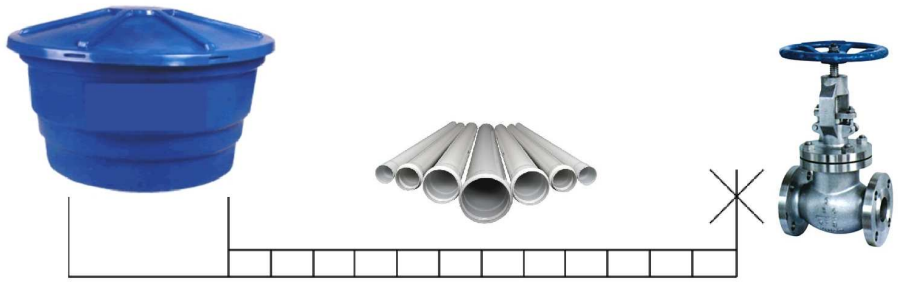

Fig. 2.1: Sistema Reservatório − Tubo − Válvula.

O sistema hidráulico escolhido pode ser facilmente observado na paisagem urbana, conforme

ilustra a figura 2.2, sendo responsável, por exemplo, pela condução de água de reservatórios para outras regiões da cidade.

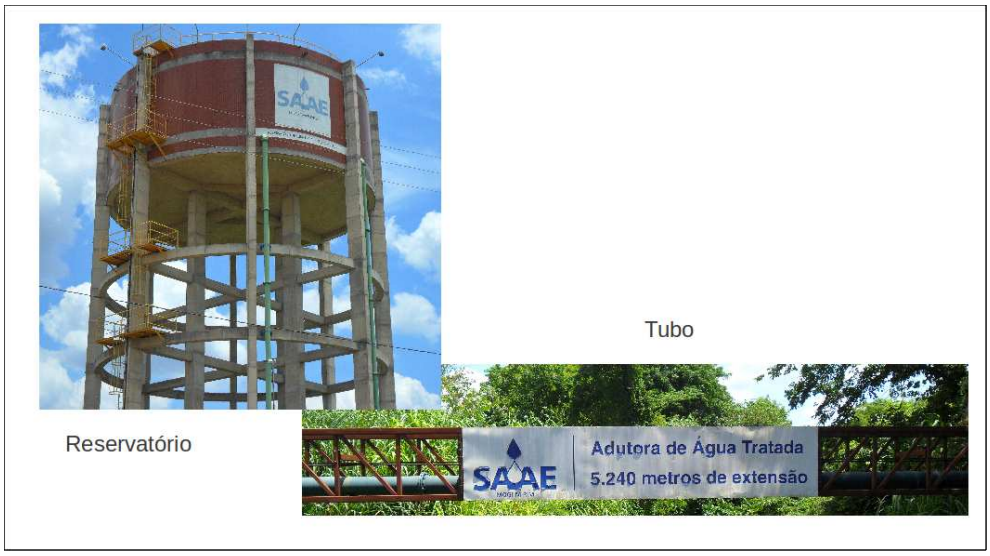

Fig. 2.2: Reservatório e tubo observados na cidade de Mogi Mirim/SP.

2.2 Transiente Hidráulico

No sistema escolhido, quando a válvula está aberta, se estabelece um fluxo contínuo de fluido no tubo, levando o sistema para o regime permanente inicial. Ao se fechar a válvula, o fluxo é interrompido e o sistema passa para o regime de transiente hidráulico. No regime de transiente hidráulico são formadas ondas de sobrepressão e subpressão internas no tubo que oscilam até o momento que a movimentação interna do fluido entra em um novo regime permanente, conhecido como regime permanente final. O transiente hidráulico é importante pois se o sistema não for corretamente dimensionado, as ondas de sobrepressão e subpressão podem levar a rompimentos ou colapsos do tubo que podem interromper a transmissão do fluido.

A figura 2.3 ilustra como o histórico da pressão são observados em dois segmentos distintos do tubo. No lado esquerdo da figura, a pressão interna do tubo é apresentada em 6 instantes de tempo. Os trechos com a cor vermelha representam ondas de sobrepressão, os trechos a cor branca representam as ondas de subpressão e os trechos com a cor azul representam a pressão igual a pressão no regime permanente inicial. Durante o transiente hidráulico, cada segmento apresenta variações de pressão,

conforme ilustrado no lado direito da figura. Essas variações vão diminuindo com o passar do tempo até o sistema atingir um novo regime permanente.

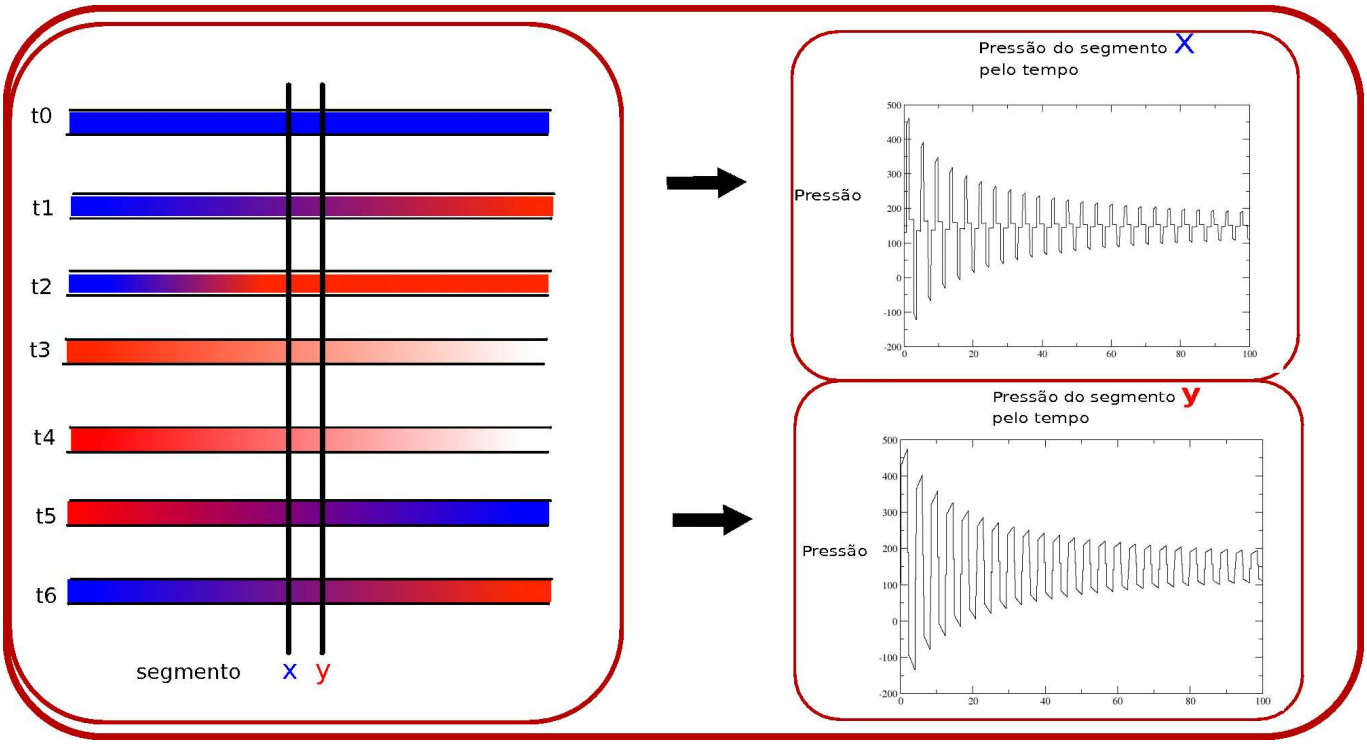

Fig. 2.3: Ilustração do histórico da Pressão nos segmentos X e Y ao longo do tempo. Ao lado esquerdo, a visualização de ondas de sobrepressão e subpressão e os dois segmentos observados. Ao lado direito, dois gráficos ilustram o histórico da pressão nos dois segmentos - X e Y - em destaque. Por ser uma ilustração, desconsidera a vaporização que ocorreria nos valores apresentados.

A descrição matemática do fenômeno do transiente hidráulico é baseada em equações diferenciais que regem o comportamente do fluido dentro do tubo. As equações da quantidade de movimento (2.1) e da continuidade (2.2) formam um par de equações diferenciais parciais hiperbólicas em termos de duas variáveis dependentes: velocidade do fluido (V) e carga piezométrica (H); e duas variáveis independentes: distância ao longo da tubulação (x) e tempo (t) [1, 16].

$$g\frac{\partial H}{\partial x} + V\frac{\partial V}{\partial x} + \frac{\partial V}{\partial t} + \frac{fV|V|}{2D} = 0 \tag{2.1}$$

$$V\frac{\partial H}{\partial x} + \frac{\partial H}{\partial t} + \frac{a^2}{g}\frac{\partial V}{\partial x} = 0 \tag{2.2}$$

onde g é a aceleração gravitacional, f é o fator de fricção do tubo, D é o diâmetro do tubo, e a é a celeridade da onda no tubo.

Estas equações podem ser simplificadas, reconhecendo-se que os termos de adveccção $V\frac{\partial V}{\partial x}$ e $V\frac{\partial H}{\partial x}$ são desprezíveis comparados aos demais termos. O resultado desta simplificação é apresentado nas equações da quantidade de movimento 2.3 e da continuidade 2.4.

$$g\frac{\partial H}{\partial x} + \frac{\partial V}{\partial t} + \frac{fV|V|}{2D} = 0 \tag{2.3}$$

$$\frac{\partial H}{\partial t} + \frac{a^2}{g}\frac{\partial V}{\partial x} = 0 \tag{2.4}$$

A solução destas duas equações fornece os valores de $H(x,t)$ e $V(x,t)$ no domínio definido por $0 < x < L$, onde L é o comprimento do tubo e e $0 < t < t_{max}$, onde t_{max} é o limite do domínio tempo.

As equações da quantidade de movimento -2.3- e da continuidade -2.4- envolvem derivadas parciais de H e V, com o respectivo x e t. No entanto, pode-se reconstruir as expressões envolvendo derivadas totais. Em geral, ambas as variáveis H e V são funções de x e t. Se a variável independente x é permitida ser uma função de t, teremos então:

$$\frac{dH}{dt} = \left(\frac{\partial H}{\partial x}\right)\left(\frac{dx}{dt}\right) + \left(\frac{\partial H}{\partial t}\right) \tag{2.5}$$

e

$$\frac{dV}{dt} = \left(\frac{\partial V}{\partial x}\right)\left(\frac{dx}{dt}\right) + \left(\frac{\partial V}{\partial t}\right) \tag{2.6}$$

Para encontrar estes valores, a equação 2.7 une as duas equações com o uso do multiplicador λ

$$g\frac{\partial H}{\partial x} + \frac{\partial V}{\partial t} + \frac{fV|V|}{2D} + \lambda(\frac{\partial H}{\partial t} + \frac{a^2}{g}\frac{\partial V}{\partial x}) = 0 \tag{2.7}$$

Qualquer valor real distinto para λ irá render duas equações de duas variáveis dependentes H e V que são equivalentes as equações 2.3 e 2.4. A seleção apropriada de dois valores particulares para λ conduz a simplificação 2.7.

A equação 2.7 expandida e feito a coleta de termos envolvendo as derivadas totais dH/dt e dV/dt produz:

$$\lambda.\left(\frac{\partial H}{\partial t} + \frac{g}{\lambda}\frac{\partial H}{\partial x}\right) + \left(\frac{\partial V}{\partial t} + \frac{\lambda a^2}{g}\frac{\partial V}{\partial x}\right) + \frac{fV|V|}{2D} = 0 \tag{2.8}$$

Se tomamos dx/dt = g/λ para dH/dt e dx/dt = $\lambda a^2/g$ para dV/dt, e se ambas as expressões para dx/dt precisam ser as mesmas, teremos:

$$\frac{\lambda a^2}{g} = \frac{g}{\lambda} \tag{2.9}$$

ou

$$\lambda = \pm\frac{g}{a} \tag{2.10}$$

Portanto,

$$\frac{dx}{dt} = \pm a. \tag{2.11}$$

Definido o valor constante a, as retas com inclinação $+a$ (referente a linha $C+$) e $-a$ (referente a linha $C-$) estabelecem às seguintes equações ordinárias

$$C+ = \begin{cases} \frac{g}{a}\frac{dH}{dt} + \frac{dV}{dt} + \frac{fV|V|}{2D} = 0 & \frac{dx}{dt} = +a \end{cases} \tag{2.12}$$

$$C- = \begin{cases} -\frac{g}{a}\frac{dH}{dt} + \frac{dV}{dt} + \frac{fV|V|}{2D} = 0 & \frac{dx}{dt} = -a \end{cases} \tag{2.13}$$

Usualmente utiliza-se a vazão Q do fluido que se relaciona com V por $Q = AV$, onde A é a área da seção transversal do tubo. Considerando-se A constante, as equações 2.12 e 2.13 são reescritas para

$$C+ = \begin{cases} \frac{dH}{dt} + B\frac{dQ}{dt} + aRQ|Q| = 0; & dx = adt \end{cases} \tag{2.14}$$

$$C- = \begin{cases} \frac{dH}{dt} - B\frac{dQ}{dt} - aRQ|Q| = 0; & dx = -adt \end{cases} \tag{2.15}$$

onde $B = \frac{a}{gA}$ e $R = \frac{f}{2gDA^2}$.

O sistema de equações diferenciais parciais originais foi reduzido para as duas equações diferenciais ordinárias ($C+$ e $C-$) que valem ao longo linhas características correspondentes (2.11), conforme ilustra figura 2.4. A resolução das equações ordinárias é geralmente feita discretizando o espaço $x \times t$, ou seja, fazendo a variável x ser múltipla de incrementos $x_i = i\Delta x$ ($i = 0, 1, \ldots, N$) e a variável t de incrementos $t_j = j\Delta t$ ($j = 0, 1, \ldots, M$). Os valores de H e Q são então obtidos numericamente em

cada ponto discretizado do espaço e do tempo, ou seja, $H(x_i, t_j)$ e $Q(x_i, t_j)$.

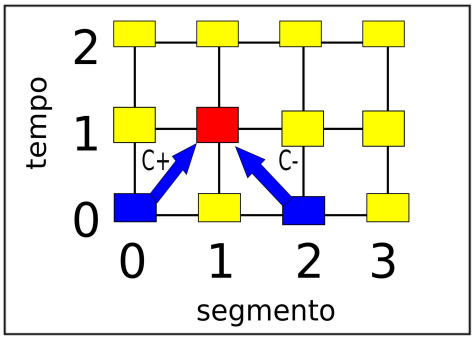

Fig. 2.4: Representação do espaço discretizado $x \times t$. Em azul, as características nos pontos anteriores (C+) e posteriores (C-) no tempo anterior (tempo 0) são utilizadas para encontrar o valor no ponto vermelho no tempo posterior (tempo 1).

Integrando as equações 2.14 e 2.15 ao longo das linhas características, temos

$$\int_{H(x_{i-1},t_{j-1})}^{H(x_i,t_j)} dH + B \int_{Q(x_{i-1},t_{j-1}}^{Q(x_i,t_j)} dQ + R \int_{x_{i-1}}^{x_i} Q|Q|dx = 0 \tag{2.16}$$

$$\int_{H(x_{i+1},t_{j-1})}^{H(x_i,t_j)} dH + B \int_{Q(x_{i+1},t_{j-1})}^{Q(x_i,t_j)} dQ - R \int_{x_{i+1}}^{x_i} Q|Q|dx = 0 \tag{2.17}$$

Usando a aproximação

$$\int_{x_{i+1}}^{x_i} Q|Q|dx = Q(x_i, t_j)|Q(x_{i+1}, t_{j-1})|\Delta x, \tag{2.18}$$

($\Delta x = x_{i+1} - x_i$)

podemos expressar $H(x_i, t_j)$ e $Q(x_i, t_j)$ como

$$Q(x_i, t_j) = \frac{C_p - C_m}{B_p + B_m} \tag{2.19}$$

$$H(x_i, t_j) = C_p - B_p Q(x_i, t_j) \tag{2.20}$$

onde

$$C_p = H(x_{i-1}, t_{j-1}) + BQ(x_{i-1}, t_{j-1}) \qquad B_p = B + R\Delta x|Q(x_{i-1}, t_{j-1})| \tag{2.21}$$

$$C_m = H(x_{i+1}, t_{j-1}) + BQ(x_{i+1}, t_{j-1}) \qquad B_m = B + R\Delta x|Q(x_{i+1}, t_{j-1})| \tag{2.22}$$

Uma solução numérica do problema transiente em um tubo irá transportar os valores de H e Q ao longo da linha de características à medida que o tempo incrementa - Δt -, conforme observado na figura 2.4.

As condições iniciais - regime permanente inicial - da pressão (H) e vazão (Q) em $t = 0$ são transportados no tempo ao longo das linhas de características no interior do domínio $x \times t$. As condições de contorno - nó juzante e nó montante - são importantes para completar a solução. Através destas condições de contorno que um regime transiente é introduzido ao sistema por meio de uma manobra hidráulica que pode ser um fechamento de uma válvula, o ligamento de uma bomba, dentre outros.

É importante ressaltar que os valores de H e Q em t_j dependem simplesmente dos valores em t_{j-1}. Além disso, uma vez conhecidos os valores de H e Q no tempo t_{j-1}, os cálculos dos N valores de H ($H(x_i, t_j)$) e Q ($Q(x_i, t_j)$) ao longo de x podem ser feitos independentemente, uma vez que cada um deles depende de dois valores calculados no tempo anterior: $H(x_{i-1}, t_{j-1})$ e $H(x_{i+1}, t_{j-1})$, e $Q(x_{i-1}, t_{j-1})$ e $Q(x_{i+1}, t_{j-1})$, respectivamente. Essa condição é crucial para que se possam aplicar estratégias de computação paralela a esse método como descreveremos a seguir.

Capítulo 3

Computação Paralela

O objetivo deste capítulo é apresentar alguns tópicos relevantes sobre computação paralela, passando pelas principais classificações de computadores paralelos. Posteriormente, discute o modelo GPU, com foco na arquitetura CUDA e o modelo OpenMP, escolhido como parâmetro de comparação. Também apresenta os conceitos de *speedup* e eficiência computacional, conceitos importantes para comparações entre códigos seriais e paralelos.

3.1 Computação Paralela

A computação paralela pode ser compreendida como "uma coleção de processadores que se comunicam e cooperam entre si para resolver rapidamente grandes problemas" [17]. Esta definição, embora contemple problemas onde a principal restrição seja processamento, ela é satisfatória para a abordagem deste trabalho. Segundo a arquitetura de *von Neumann*, um computador pode ser entendido por um processador (CPU), um sistema de memória e um sistema de entrada e saída (*I/O*) [18]. Esta definição, embora generalista, auxilia na explicação de como se dá o funcionamento dos computadores atuais, que se inspiraram nesta arquitetura. Nesta arquitetura (figura 3.1), o funcionamento dá-se por ciclos de execução onde o processador (CPU) busca uma instrução da memória, descodifica a instrução, busca os dados necessários para processar a instrução e executa a instrução.

Um computador paralelo pode ser entendido como combinações de máquinas de *von Neumann* trabalhando juntas para um mesmo objetivo computacional. O método de interconexão entre os com-

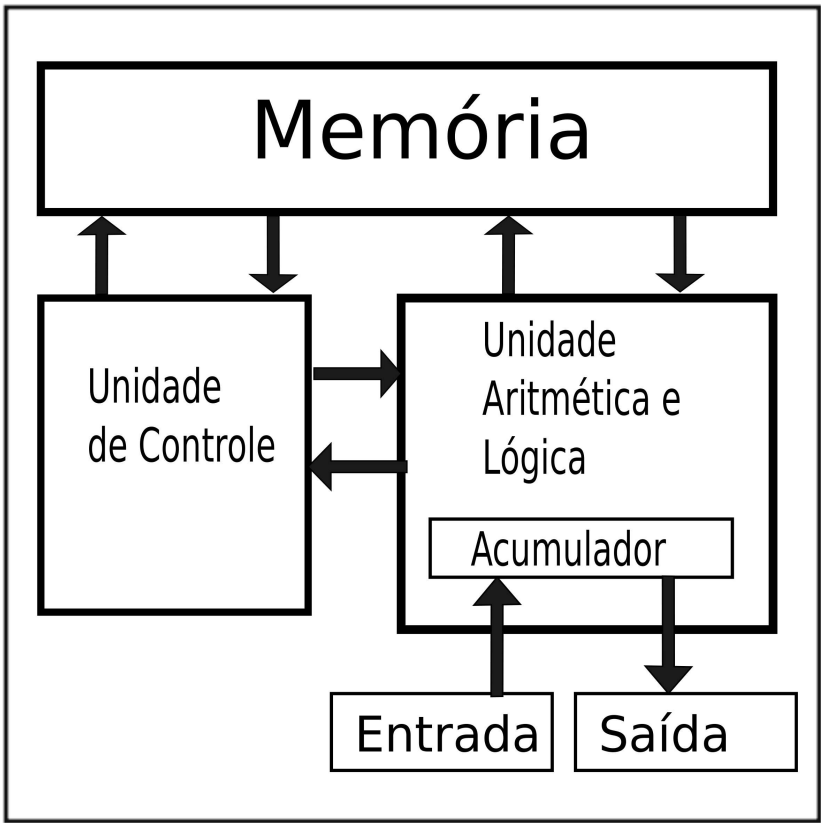

Fig. 3.1: Visão geral da arquitetura de Von Neumann (adaptado de [18])

putadores, número de unidades e outros aspectos técnicos é o que difere uma arquitetura de outra.

Para Flynn [19], as arquiteturas e os computadores paralelos podem ser classificados quanto ao fluxo de instruções e quanto ao fluxo de dados. As arquiteturas com um único fluxo de dados e um único fluxo de instruções são classificados como arquitetura SISD (*Single Instruction, Single Data*). Esta arquitetura é uma abstração da máquina de Von Neumann. Arquitetura com uma única instrução, mas com paralelismo nos dados são classificados como SIMD (*Single Instruction, Multiple Data*). Pela classificação de Flynn, a arquitetura CUDA se encontra nesta categoria. Já arquiteturas com múltiplas instruções funcionando ao mesmo tempo, que trabalham em um mesmo conjunto de dados são classificadas como MISD (*Multiple Instruction, Single Data*). Não há representantes nesta categoria de arquitetura. E as arquiteturas com Múltiplas instruções funcionam ao mesmo tempo, que trabalham em um múltiplos conjuntos de dados são classificadas como MIMD (*Multiple Instruction, Multiple Data*). Nesta categoria se encontram-se cluster do tipo Beowulf e computadores massivamente paralelos.

Quanto ao sistema de memória, Tanembaum [18] apresenta dois padrões: sistema com memória compartilhada (figura 3.2) onde a memória do sistema é diretamente acessível por todos os processadores e sistemas de memória distribuída (figura 3.3), onde cada processador tem sua própria memória.

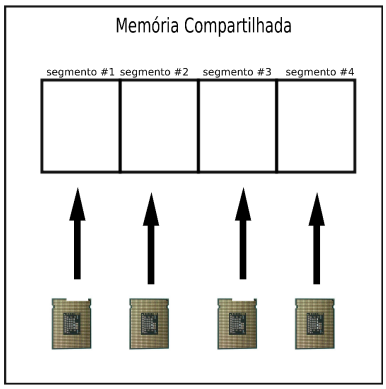

Fig. 3.2: Visão geral de um sistema de memória compartilhada (adaptado de [18])

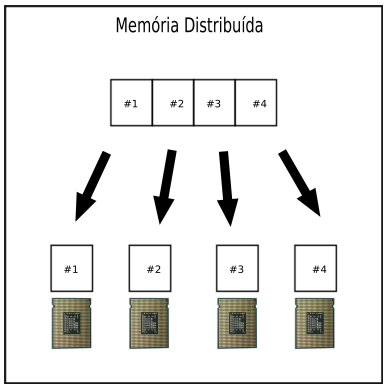

Fig. 3.3: Visão geral de um sistema de memória distribuída (adaptado de [18])

Para Tanembaum, [18] as técnicas de computação paralela podem ser classificadas quanto a unidade de processamento em fracamente ou fortemente acopladas. Computadores fracamente acoplados possuem baixa largura de banda, alta taxas de atraso para troca de mensagens e podem não estar fisicamente no mesmo computador. Nesta categoria, destacamos os computadores que utilizam o padrão MPI *(Message Passing Interface)* [20]. Neste padrão, uma aplicação é constituída por programas em execução (processo) que se comunica por meio do envio e recebimento de mensagens (blocos de dados) para outros processos. Diferentes problemas na área de engenharia civil apresentam ganhos computacionais ao usar paralelismo baseado em MPI: Wright e Villanueva [21] e Teng [22] apresentaram ganhos em simulações sob inundação, trabalhando com cenários distintos. Agrawal1 e Mathew [23] utilizaram duas técnicas de paralelismo fracamente acoplado com algoritmo genético para solução de problemas envolvendo roteamento de redes de trânsito. Goring e Walters [24] avaliam os ganhos computacionais da troca de mensagens comparado a memória compartilhada no modelo da equação de onda. Por outro lado, computadores fortemente acoplados possuem alta largura de banda com baixo atraso para troca de mensagens entre as unidades de processamento. Nesta categoria, destacamos a arquitetura GPU, foco deste trabalho e a *API* OpenMP, utilizada neste trabalho como parâmetro de comparação.

3.2 Arquitetura GPU

Da perspectiva do programador, a arquitetura GPU consiste de uma máquina tradicional com uma ou mais CPUs (*host*), e uma ou várias placas de vídeo (*devices*), equipados com processadores paralelos com várias unidades aritméticas de execução (figura 3.4). O *device* atua como um co-processador que executa em paralelo com o *host*.

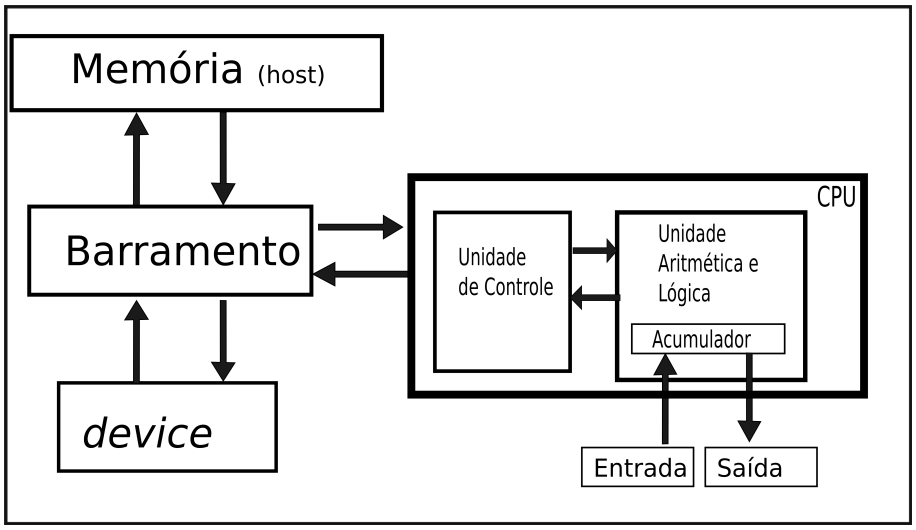

Fig. 3.4: Visão geral da arquitetura GPU.

Segundo Verdière [7], existem três empresas no mercado que provêm este tipo de arquitetura: NVidia, Intel e AMD. A NVIdia foi a pioneira no uso de processamento gráfico para computação de propósito geral, e hoje é líder de mercado. Por isto, foi a arquitetura escolhida neste trabalho. A arquitetura CUDA, criada pela empresa NVidia é um conjunto *hardware* e *software*, que permite ao programador explorar os recursos computacionais das placas em aplicações de uso geral.

Ao se escolher a arquitetura CUDA é importante verificar se o *device* é compatível e qual a sua capacidade computacional(*capability*). *Capability* é definido por um número de revisão principal e secundário. Os dispositivos com o mesmo número de revisão principal são da mesma arquitetura do núcleo. O número de revisão secundário corresponde a uma melhoria incremental à arquitetura do núcleo, incluindo possivelmente características novas. As limitações da arquitetura e da placa adquirida pode ser verificada ao se consultar as características da *capability* [25].

A arquitetura CUDA se apresenta promissora para algoritmos que possam ser classificados como problemas de uma única instrução e múltiplos dados (SIMD). Isto se deve a sua organização interna

e a relação hierarquizada de alguns componentes.

Na arquitetura CUDA, um *kernel* pode ser entendido como uma chamada a uma função ou rotina -diferentemente de outras rotinas que executam na CPU, esta função especial irá ser executada no GPU (*device*).[26] No *kernel*, os trechos de execução que são executados de forma paralela, em que cada segmento atua em um fluxo de dados diferente. As *threads* são organizadas em blocos. Um bloco é um arranjo de *threads*, organizado em 1D, 2D ou 3D. Cada bloco possui uma organização 2D, dimensionado em 5 por 3 por 1, totalizando-se 15 *threads* por bloco. [25] [26] (fig. 3.5) Por sua vez, os blocos são organizados em *grids*. *Grids* podem ser definidas como um arranjo 1D ou 2D de blocos. Na figura 3.5, cada *grid* possui uma organização 2D, dimensionado em 3 por 2 blocos, totalizando-se 6 blocos presentes na *grid* do primeiro *kernel*[26].

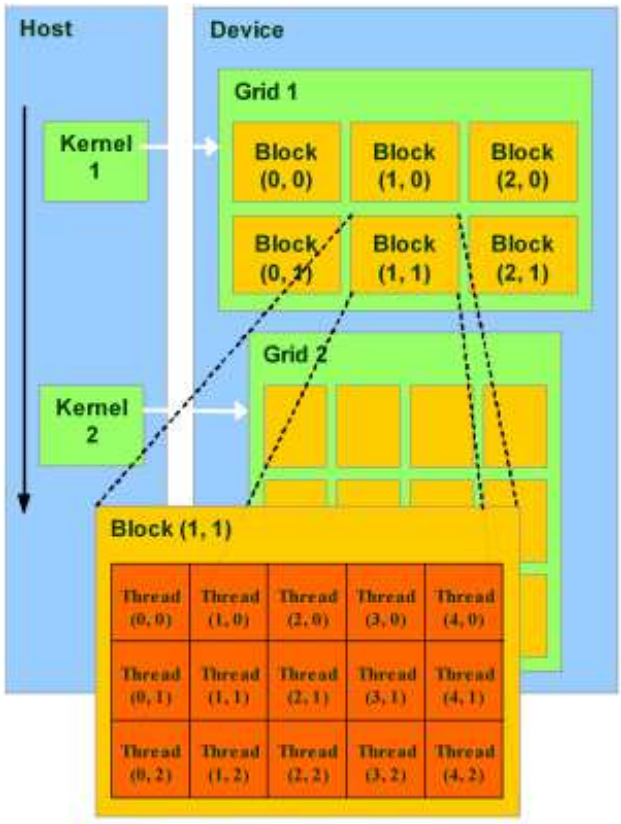

Fig. 3.5: Exemplo de *thread*, bloco, *grid* e *kernel*. Adaptado de [27].

Tanembaum [28] explica a relação entre custo, desempenho e preço da memória. Quanto mais rápido o acesso a memória, maior será o seu custo e menor o seu tamanho. Este conceito conhecimento como hierarquia de memória também se aplica aos tipos de memória existentes no *device*, conforme

ilustra figura 3.6.

Nos *devices* com *capability* até 1.x, a memória global permite leitura e gravação por *grid*, de forma lenta e não cacheável (*uncached*). Esta memória requer sequênciamento e alinhamento a cada bloco de 16 bytes. Em *devices* com *capability* 2.x, esta memória é lenta, mas cacheável. A memória global é alocada no *device* e acessada através de transações de 32-, 64-, ou 128 bytes. É acessível por todas as linhas de execução do *grid*, e gerenciada pelo *host* via funções específicas usadas para alocar, desalocar e transferir dados do *host* para o *device* e do *device* para o *host* . A memória textura (*texture memory*) possui cache otimizado para padrão de acesso espacial 2D, sendo adequado a aplicações para manipulação de imagens. A memória constante (*constant memory*), é adequada para o armazenamento de os valores constantes e argumentos de *kernel*. É uma memória com cache de 8 kilobytes. A memória compartilhada (*shared memory*) é uma memória, acessível por todas as linhas de execução *threads* do mesmo bloco, e sua função é auxiliar a redução de latência de acesso a memória global. Nesta memória, podem ocorrer situações de bloqueio, ou seja, situações bloqueantes entre duas ou mais *threads*. Os registradores (*registers*) são memórias locais a cada linha de execução, e a quantidade de memória é definida pelo compilador. São memórias rápidas e pequenas. A memória local (*local memory*) é lenta, comparado a memória dos registradores, e não cacheável. É utilizado para todas as situações de escopo local que não se extrapolam o limite da memória registradores.

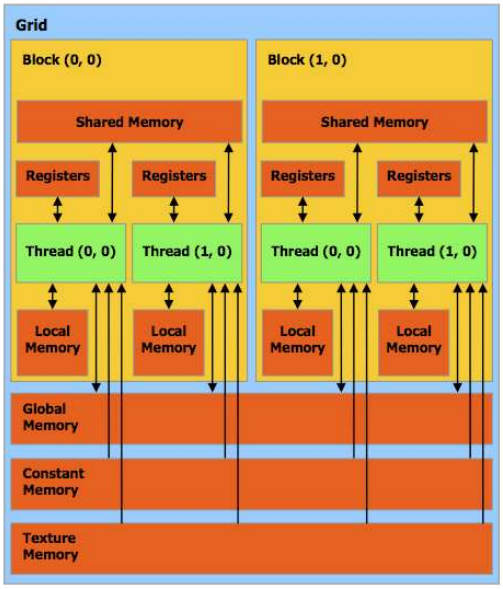

Fig. 3.6: Hierarquia de memórias em CUDA (Adaptado de [25]).

A arquitetura CUDA inclui um ambiente de software com suporte a diversas linguagens: Fortran, OpenCL, C, openACC, dentre outras. A linguagem escolhida neste trabalho foi a linguagem CUDA-C. A linguagem CUDA-C permite aos programadores usem C como um linguagem de programação de alto nível no padrão C-ANSI estendido.

CUDA C permite definir funções em C, denominadas *kernel* que, quando chamadas, são executadas paralelamente por N *threads* diferentes do *device*. Na figura 3.7, o fluxo de execução é desviado para GPU ao chamar a função VecAdd, com uso de 100 *threads* em 20 blocos.

```
float* h_A,h_B, h_C,d_A,d_B,d_C;
// Código em device
__global__ void VecAdd(const float* A, const float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

int main (int argc, char ** argv)
{
    int N = 2000;
    size_t size = N * sizeof(float);

    h_A = (float*)malloc(size);
    h_B = (float*)malloc(size);
    h_C = (float*)malloc(size);

 // Alocação dinâmica no device
    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

 // Copiar vetores do host para o device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    VecAdd<<<20, 100>>>(d_A, d_B, d_C, N);
 // Copiar vetor do device para o host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
}
```

Fig. 3.7: Exemplo de função em C com uso de CUDA-C. Neste exemplo, aloca-se três vetores de 2000 posições em *host* e três vetores em *device* OS vetores A e B são copiados do *host* para *device*, a soma é calculadas via *kernel* com uso de 20 blocos e 100 *threads* em cada bloco. O resultado da soma, alocado no vetor C, copiado do *device* para *host*.
.

Por abstrair do programador algumas complexidades do paralelismo, a curva de aprendizado, ou seja, o tempo necessário para o aprendizado do OpenMP mostrou-se bem menor comparado a GPU. Pela característica de memória compartilhada, fortemente acoplado, e relativa facilidade no aprendizado, esta *API* foi escolhida para se comparar os resultados da arquitetura CUDA.

Para Ikeda [29], ao se comparar CPU e GPU, existem alguns desafios e abordagens: O núcleo da CPU foi projetado para executar, à velocidade máxima, uma única linha de execução com instruções sequenciais, enquanto a GPU tem um projeto elaborado para executar milhares de linhas de execução em paralelo. A maior parte dos transistores da GPU trabalha no processamento do dado em si. Já os transistores da CPU trabalham com o intuito de melhorar o desempenho da execução sequencial.

A CPU é capaz de executar uma ou duas linhas de execução por núcleo e a mudança de uma linha de execução para outra custa centenas de ciclos. Por outro lado, a GPU pode manter até 1024 linhas de execução por multiprocessador além de tipicamente fazer trocas de linha de execução a cada ciclo. Aplicações com muitas operações aritméticas e altamente paralelizáveis podem se beneficiar da GPU pois a independência entre as *threads* permite que pacotes múltiplos de dados sejam processados em uma única solicitação.

OpenMP é uma *API* baseada no modelo de programação paralela de memória compartilhada para arquiteturas de múltiplos processadores. Da perspectiva do programador, a API OpenMP consiste de uma máquina tradicional, e seu código utiliza diretivas de compilação que utiliza todos os processadores disponíveis (figura 3.8).

Todo paralelismo do OpenMP é especificado através de três componentes básicos. O primeiro componente são as diretivas de compilação, ou seja, comandos no código-fonte não compilados, dirigidos ao pré-processador e iniciados com o caracter sustenido (#). O segundo componente básico é a biblioteca de execução. O computador que executar o código OpenMP precisa ter a biblioteca OpenMP previamente instalada. No código C apresentado na figura 3.8, a biblioteca é invocada na primeira linha, (#include <omp.h>). O terceiro componente básico são as variáveis de ambiente, que permite, dentre outras configurações, definir o número de núcleos que se espera ocupar ao executar o código paralelo. Caso este componente não esteja presente, a biblioteca OpenMP define como padrão de núcleos o número de núcleos presentes no *hardware*.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main (int argc, char **argv)
{
    int i;
    int chunk = 5;
    int N = 2000;
    size_t size = N * sizeof(float);

    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    float* h_C = (float*)malloc(size);

// Diretivas de compilação onde inicaliza trecho paralelo.
    #pragma omp parallel shared(h_A,h_B,h_C,chunk) private(i)
    {
#pragma omp for schedule(dynamic,chunk) nowait
        for(i = 0; i< N; i++) {
            h_C[i] = h_A[i] + h_B[i];
        }
    }
}
```

Fig. 3.8: Exemplo de código C com uso de OpenMP. Neste exemplo, aloca-se três vetores de 2000 posições. O vetor A, B, C e a variável chunk são compartilhadas por todas as *threads*.

3.3 Métricas para análise de desempenho

Para Kan [30], as métricas de *software* podem ser classificadas em três categorias: métricas de produto, métricas de processo e métricas de projeto. As métricas de produto descrevem as características do produto como tamanho, complexidade e desempenho. Neste trabalho, a métrica de produto foi a de desempenho.

Para Jain [31], alguns erros comuns devem ser evitados ao se comparar o desempenho computacional de dois sistemas. Os erros mais comuns são: não ter objetivos bem definidos ou tê-los de forma tendenciosa, análise sem entendimento do problema, omissão de suposições e limitações do sistemas e não conhecimento da carga de trabalho do algoritmo. Jain [31], Foster [32] e Grama [33] citam como duas principais métricas de desempenho o *speedup* e a eficiência computacional. *Speedup* refere-se o quanto um algoritmo paralelo é mais rápido que um algoritmo sequencial correspondente [33]. Esta grandeza é mensurada conforme a equação 3.1.

$$S_p = \frac{T_1}{T_p} \tag{3.1}$$

Assim, *speedup* é a razão entre o tempo de execução em um único processador (T_1) pelo tempo de execução em multiprocessadores, com uso de p processadores (T_p). Para Foster [32], o tempo de execução nem sempre é a métrica mais conveniente pelo qual se avalia o desempenho do algoritmo paralelo. Como o tempo de execução tende a variar com o tamanho do problema, tempos de execução deve ser normalizados quando comparando o desempenho do algoritmo em tamanhos diferentes do problema.

A equação 3.2 apresenta a equação da eficiência, ou seja, a fração de tempo que processadores gastam na realização de trabalho útil. O autor enfatiza que esta é uma métrica relacionada que às vezes pode fornecer uma medida mais conveniente de qualidade do algoritmo paralelo.

$$E_p = \frac{T_1}{T_p * p} \tag{3.2}$$

A equação da eficiência é a razão entre o tempo de execução com um processador (T_1) pelo tempo de execução usando multiprocessador multiplicado pelo número de processadores disponíveis ($T_p * p$) A eficiência máxima é obtida quando todos os processadores são utilizados o tempo todo e o *speedup* é *p*. Esta situação de eficiência máxima é denominado *speedup* linear.

Após conhecer as principais características da arquitetura CUDA e justificar a escolha da *API* OpenMP como parâmetro de comparação, os próximos capítulos apresentam as estratégias de paralelismo na resolução do problema proposto e os respectivos resultados.

Capítulo 4

Estratégias de paralelismo

Este capítulo apresenta o código serial, bem como a estratégia do código paralelo implementado com a *API OpenMP* e as estratégias do código paralelo implementado com uso da arquitetura CUDA.

4.1 Desenvolvimento Serial

A implementação serial do modelo foi escrita na linguagem C e o pseudo-código da implementação serial é apresentado na figura 4.2. Conforme apresentado, no passo 1, os parâmetros de entrada são recebidos via arquivo texto. Estes parâmetros são: altura do reservatório (hr), fator de fricção *Darcy-Weisbach* (f), largura do tubo (l), diâmetro do tubo (d), número de segmentos do tubo (n), tempo máximo de simulação (tmax), celeridade da onda (a) e o coeficiente de descarga vezes a área da vávula (cdao).

No exemplo do arquivo de entrada ilustrado na figura 4.1, a altura do reservatório é de 150 metros, o fator de fricção é um parâmetro adimensional que é utilizado para calcular a perda de carga em uma tubulação devida ao atrito. No exemplo, este fator é de 0.035. A largura do tubo (l) ilustrado é de 1200 metros e seu diâmetro é de 30 centímetros. Este tubo é dividido em 10 segmentos (n) de 120 metros entre um segmento e outro. O tempo máximo de simulação (tmax) apresentado no arquivo de entrada é de 100 segundos. Após decorrido o tempo máximo, não se pode afirmar que o sistema esteja em um regime permanente final, porém as oscilações internas na tubulação causadas pelo transiente irão apresentar uma amplitude menor de pressão e vazão. Por fim, a celeridade da onda (a) trata da

velocidade com que uma onda de pressão se propaga num fluido é dado em metros por segundo e o coeficiente de descarga vezes a área da vávula (cdao) é passado em metros quadrados.

```
hr      150
f       0.035
l       1200
d       0.3
n       10
tmax    100
a       1200
cdao    0.0009
```

Fig. 4.1: Exemplo do arquivo de entrada

Após a leitura dos parâmetros de entrada, o próximo passo é encontrar os valores das constantes dx, que representa o tamanho do segmento a ser calculado e dt que representa o intervalo entre um tempo (Δt) e outro. Neste trecho fica evidente como as variáveis número de segmentos (n), celeridade da onda (a) e tamanho do tubo (l) possuem impacto direto no tamanho das malhas computacionais, criadas dinamicamente.

A criação dinâmica das malhas computacionais, ou seja, uma matriz dinâmica para pressão (H) e outra para vazão (Q) é o procedimento realizado no passo 3. Nesta etapa, dois ponteiro de números ponto flutuante (float*) são declarados para alocação dinâmica das matrizes, conforme apresentado abaixo.

```
...
size_t sizeMATRIX = N * sizeof(float);
size_t sizeVECTOR = ns * sizeof(float);
...
HMatriz = (float*)malloc(sizeMATRIX);
QMatriz = (float*)malloc(sizeMATRIX);
```

Durante o desenvolvimento da versão serial, uma estrutura de ponteiro de ponteiro de números ponto flutuante (float**) chegou a ser desenvolvido. Porém, com a complexidade em se alocar vetor de ponteiros de números flutuantes e se utilizar um laço de repetição para alocação dinâmica de cada ponteiro de ponto flutuante, optou-se em manter-se um único vetor de ponto flutuante e uma variável de índice (ns) para cada variação em Δt. A variável de índice (ns) representa o número de segmentos, acrescido do nó de montante e o nó de jusante. (ns = n + 2)

```
1) Leitura dos parâmetros de entrada, via arquivo de entrada .dat
        altura do reservatório                  (hr)
        fator de fricção Darcy-Weisbach         (f)
        largura do tubo                         (l)
        diâmetro do tubo                        (d)
        número de segmentos do tubo             (n)
        tempo máximo de simulação               (tmax)
        celeridade da onda                      (a)
        área de seção transversal da válvula    (cdao)

2) Cálculo das constantes
        dx = l/n
        dt = dx/a

3) Alocação dinâmica das matrizes H (dx,dt) e Q (dx,dt)

4) Cálculo do Regime Permanente Inicial
        Para x=0, x<=n, X++ {
                calcular H[x]
                calcular Q[x]
        }

5) Enquanto t<=tmax {
        t = t + dt
        Para x=0, x<=n, X++ {
                Calcular CP, BP, CM, BM ( baseado no dt anterior)

                Q[x] = (CP - CM) / (BP + BM )
                H[x] = CP - BP * Q[x]
        }
}
```

Fig. 4.2: Pseudo-código da implementação serial.

O passo 4 refere-se ao cálculo do Regime Permanente Inicial, ou seja, o cálculo das características no tempo inicial ($t = 0$) em todos os segmentos do tubo no início da simulação. Na estrutura desenvolvida no passo 3, o cálculo é realizado em um laço de repetição e os valores atualizados nas primeiras posições dos vetores HMatriz, QMatriz e TMatriz, conforme destacado no trecho abaixo:

```
t = 0;

qo = sqrt(((cdao*cdao)*2*hr*hr)/(1+(cdao*cdao)*2*g*n*r));

for(i=0; i<= ns ; i++) {
```

```
        H[i] = hr-(i)*r*qo*qo;
        Q[i] = qo;
        HMatriz[i] = H[i];
        QMatriz[i] = Q[i];
        TMatriz[i] = t;
    }
```

O passo 5 refere-se ao cálculo do Regime Transiente, onde se calcula a pressão e a vazão baseadas nas características no tempo anterior e nos segmentos vizinhos. Na figura 4.3, observa-se em vermelho o ponto no segmento 1 no tempo $t = 1$ e as características necessárias para seu cálculo em azul: a característica no segmento 0 e 2 do tempo $t = 0$.

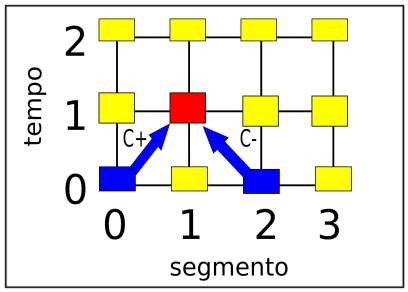

Fig. 4.3: Representação das características necessárias dentro da malha computacional.

Após o cálculo de todos os elementos da matriz pressão (H) e vazão (Q), tem-se o histórico esperado. Ao se coletar uma coluna referente a determinado segmento, tem-se o histórico do transiente neste segmento. A figura 4.4 apresenta a evolução computacional no passo 5. Neste exemplo, a malha computacional com dez segmentos e oito tempos (Δts). Em 4.4(a), o segmento 3 no tempo 1 é calculado, utilizando-se a informação no tempo anterior no segmento 2 ($C+$) e utilizando-se a informação no tempo anterior no segmento 4 ($C-$). Em 4.4(b), o segmento 4 no tempo 1 é calculado, utilizando-se dados do segmento 3 no tempo 0 ($C+$) e do segmento 5 no tempo 0 ($C-$). Em 4.4(c), o segmento 5 no tempo 1 é calculado, utilizando-se dados do segmento 4 no tempo 0 ($C+$) e do segmento 6 no tempo 0 ($C-$). Em 4.4(d), o segmento 6 no tempo 1 é calculado, utilizando-se dados do segmento 5 no tempo 0 ($C+$) e do segmento 7 no tempo 0 ($C-$). A etapa do passo 5 está incluido em um laço de repetição, conforme apresentado abaixo.

```
while(t < tmax) {
   t=t+dt;
   controle++;
      /*  Pontos Interiores  */
      for(i=1 ; i<ns ; i++){
         cp = H[i-1]+b*Q[i-1];
         bp = b+r*fabs(Q[i-1]);
         cm = H[i+1]-b*Q[i+1];
         bm = b+r*fabs(Q[i+1]);
         qp[i] = (cp-cm)/(bp+bm);
         hp[i] = cp-bp*qp[i];
      }

      /*  Condicao de Contorno de montante: Reservatorio  */
      hp[0] = hr;
      cm = H[1]-b*Q[1];
      bm = b+r*fabs(Q[1]);
      qp[0] = (hr-cm)/bm;

      /*  Condicao de contorno de jusante: Válvula  */
      cp = H[ns-2]+b*Q[ns-2];
      qp[ns-1] = 0;
      hp[ns-1] = cp;

      /*  Atualizacao das variaveis  */
      for(i=0 ; i<ns ; i++) {
         H[i] = hp[i];
         Q[i] = qp[i];
      }
      /*  Atualizacao na matriz  */
      for(i=0 ; i<=ns ; i++) {
         HMatriz[(controle * ns ) + i] = H[i];
         QMatriz[(controle * ns ) + i] = Q[i];
         TMatriz[(controle * ns ) + i] = t;
      }
```

```
}
```

O código fonte da implementação serial, evidenciado acima apresenta uma única instrução (cálculo de CP,CM,BP e BM) em diferentes pontos das matrizes pressão e vazão. Por isto, o passo 5 apresentado é o passo mais adequado a ser implementado com o uso de arquiteturas paralelas SIMD.

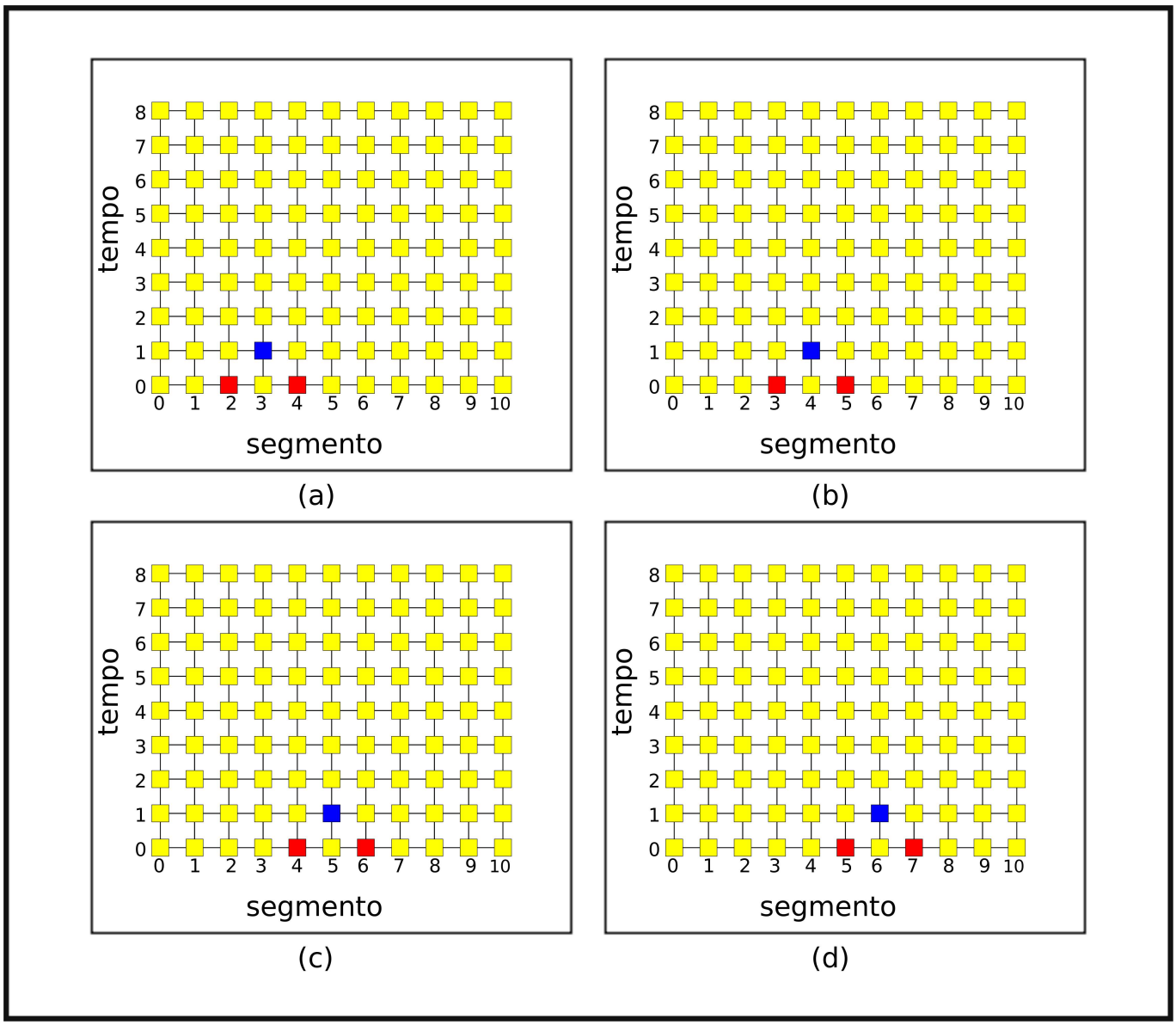

Fig. 4.4: Implementação Serial conforme computação ocorre: Em (a) ocorre o cálculo da pressão e vazão no segmento 3 no tempo 1 com uso da pressão e vazão do tempo 0 no segmento 2 (C+) e no segmento 4 (C-). Os segmentos 4, 5, 6 no tempo 1 são calculados respectivamente em (b), (c) e (d).

4.2 Desenvolvimento Paralelo

A implementação com uso da *API OpenMP* faz uso de três componentes básicos: diretivas de compilação, variáveis de ambiente e bibliotecas de execução. O tempo necessário para o aprendizado desta API mostrou-se bem menor, comparado ao tempo necessário para o aprendizado da arquitetura CUDA. O principal fator observado para que isto ocorra é o encapsulamento de algumas complexidades no desenvolvimento de códigos paralelos (como ponto de sincronização, por exemplo).

Na implementação paralela com *OpenMP*, foram paralelizados os passos 4 e 5 da figura 4.2. O trecho de código referente ao passo 4 pode ser observado abaixo. Neste trecho, todas as variáveis são compartilhadas entre todas as *threads*, exceto a variável *i*, declarada como privativas (*private*).

```
t = 0;
qo=sqrt(((cdao*cdao)*2*hr*hr)/(1+(cdao*cdao)*2*g*n*r));

#pragma omp parallel shared(H,Q,HMatriz,QMatriz,TMatriz,hr,r,qo,t,ns,chunk) private(i)
{
     #pragma omp for schedule(dynamic,chunk) nowait
          for(i=0; i<= ns ; i++) {
               H[i] = hr-(i)*r*qo*qo;
               Q[i] = qo;
               HMatriz[i] = H[i];
               QMatriz[i] = Q[i];
               TMatriz[i] = t;
          }
}
```

Nesta implementação, o quinto passo da figura 4.2 possui trechos paralelos no cálculo dos segmentos internos, atualização das variáveis e atualização das matrizes, conforme destacado a seguir.

```
while(t <= tmax) {
     t=t+dt;
     controle++;

          /*  Pontos Interiores  */

          #pragma omp parallel shared(H,Q,hp,qp,r,b,ns,chunk) private(i,cm,bm,cp,bp)
          {
```

```
                #pragma omp for schedule(dynamic,chunk) nowait
                for(i=1 ; i<ns ; i++){
                        cp = H[i-1]+b*Q[i-1];
                        bp = b+r*fabs(Q[i-1]);
                        cm = H[i+1]-b*Q[i+1];
                        bm = b+r*fabs(Q[i+1]);
                        qp[i] = (cp-cm)/(bp+bm);
                        hp[i] = cp-bp*qp[i];
                }
        }

        ...

        /* Atualizacao na matriz */

        #pragma omp parallel shared(H,Q,HMatriz,QMatriz,TMatriz,ns,t,chunk) private(i)
        {
                #pragma omp for schedule(dynamic,chunk) nowait
                        for(i=0 ; i<=ns ; i++) {
                                HMatriz[(controle * ns ) + i] = H[i];
                                QMatriz[(controle * ns ) + i] = Q[i];
                                TMatriz[(controle * ns ) + i] = t;
                        }
                }
         } // Fim do laço for
} // Fim do laço while
```

A figura 4.5 apresenta a evolução computacional no passo 5. Esta figura representa a execução OpenMP com uso de duas *threads*. A *thread* 1 é representado pela cor vermelha enquanto a *thread* 2 é representado pela cor verde. No tempo computacional (a), a primeira *thread* calcula o segmento 1 no tempo 1, enquanto a segunda *thread* realiza o mesmo cálculo no segmento 2. Em destaque, nos segmentos 0 e 2 no tempo anterior (0) estão $C+$ e $C-$ utilizados pela *thread* 1, e nos segmentos 1 e 3 no tempo 0 encontram-se $C+$ e $C-$ utilizados pela *thread* 2. No tempo computacional seguinte (figura 4.5 (b)), os segmentos 1 e 2 já estão calculados, e as *threads* 1 e 2 realizam o mesmo trabalho, porém nos segmentos 3 e 4, respectivamente. O tempo computacional seguinte (figura 4.5 (c)), o mesmo procedimento para os segmentos 5 e 6, e assim por diante, até o cálculo completo de todos os elementos da Pressão (H) e Vazão (Q) do sistema.

A implementação paralela com uso da arquitetura CUDA realiza as alocações dinâmicas de me-

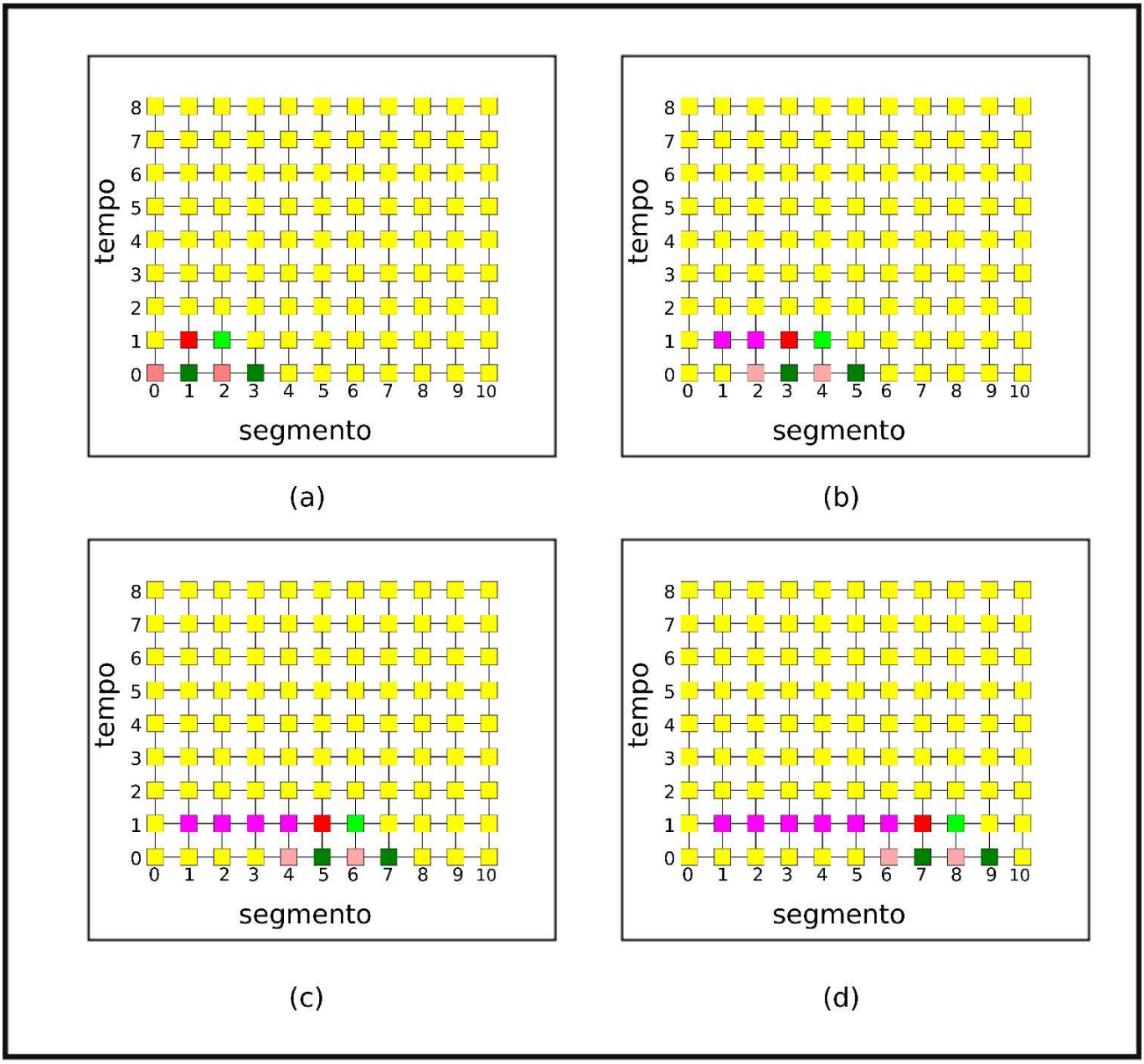

Fig. 4.5: Implementação paralela com a API OpenMP, utilizando-se duas threads: Em (a), duas threads calculam os segmentos 1 e 2 no tempo 1. A primeira thread, no segmento 1 utiliza C+ do segmento 0 no tempo 0 e C- do segmento 2 no tempo anterior. A segunda thread, no segmento 2 utiliza C+ do segmento 1 no tempo 0 e C- do segmento 3 no tempo anterior. Em (b), as duas threads calculam os segmentos 3 e 4 no tempo 1. Em (c), duas threads calculam os segmentos 5 e 6. Em (d), os segmentos 7 e 8 são calculados paralelamente. Os segmentos calculados são marcados em (b), (c) e (d) com a cor rosa.

mória em RAM (malloc) e a alocação de memória nas placas (cudaMalloc). Os resultados dos cálcu-

los realizados em *device* ficam armazenados na memória global da placa até serem transferidos para a memória RAM (cudaMemcpy), conforme evidencia o trecho a seguir.

```
size_t sizeMATRIX = N * sizeof(float);
size_t sizeVECTOR = ns * sizeof(float);

HMatriz = (float*)malloc(sizeMATRIX);
QMatriz = (float*)malloc(sizeMATRIX);
TMatriz = (float*)malloc(sizeMATRIX);

cutilSafeCall( cudaMalloc((void**)&d_HMatriz, sizeMATRIX) );
cutilSafeCall( cudaMalloc((void**)&d_QMatriz, sizeMATRIX) );
cutilSafeCall( cudaMalloc((void**)&d_TMatriz, sizeMATRIX) );


...

calc_HQ_RegimeTransiente<<<nblocks, nthreads>>>(d_HMatriz, d_QMatriz, d_TMatriz);

cudaMemcpy(TMatriz , d_TMatriz, sizeMATRIX, cudaMemcpyDeviceToHost);
cudaMemcpy(QMatriz , d_QMatriz, sizeMATRIX, cudaMemcpyDeviceToHost);
cudaMemcpy(HMatriz , d_HMatriz, sizeMATRIX, cudaMemcpyDeviceToHost);

...
```

Durante o desenvolvimento com uso da arquitetura CUDA, duas implementações foram desenvolvidas, focando-se nos passos 4 e 5 da figura 4.2. Na primeira implementação ilustrado na figura 4.6, um único bloco com múltiplas *threads* é invocado pelo *kernel* - função que executa no *device* -.

A figura 4.6 (a) apresenta o cálculo do tempo 1, enquanto em (b), é apresentado o cálculo do tempo 2. Esta implementação apresentou ganhos computacionais expressivos, porém apresentou dois problemas. O primeiro problema refere-se a confiabilidade dos resultados. Para Tanembaum [18], "a taxa que o processo realiza sua computação não é uniforme e provavelmente não reproduzível". Isto significa a possibilidade de uma *thread* executar mais rápida que outra, e solicitar a informação $C+$ ou $C-$ antes que segunda *thread* a tenha calculado. Este erro ocorreu nesta implementação em condições específicas, normalmente quando solicitado um grande número de *threads* por bloco.

O segundo problema se refere ao tamanho do tubo. Como um único bloco é invocado pelo *kernel*, para que cada *thread* calcule um segmento apenas, as simulações seriam limitadas a tubos com o

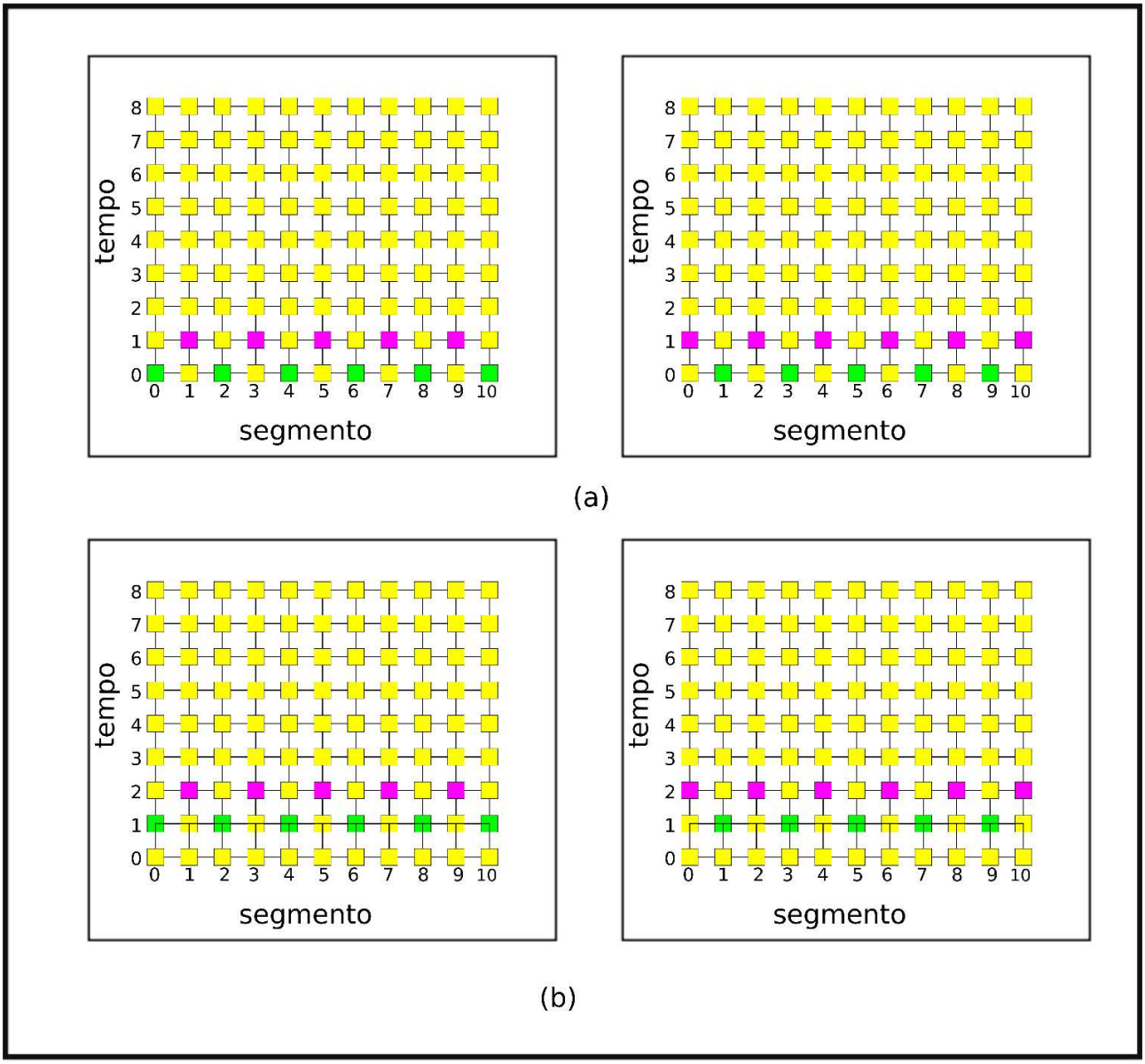

Fig. 4.6: Primeira implementação com uso da arquitetura CUDA. Em (a), as threads calculam os segmentos impares no tempo 1 (esquerda) e posteriormente os segmentos pares tempo 1 (direita). Em (b) as threads calculam os segmentos impares no tempo 2 (esquerda) e posteriormente os segmentos pares tempo 2.

número de segmentos menores ao número máximo de *threads* disponíveis na placa. Quanto o tamanho do tubo ultrapassar o número de *threads* disponíveis na placa, tornaria-se necessário escalonar a quantidade de segmentos cada *threads* seria responsável em calcular. O desenvolvimento de um escalonador de segmentos chegou a ser feito. Porém, com o crescimento da complexidade do código, e pelo problema de confiabilidade, optou-se pelo desenvolvimento com uso de uma nova estratégia.

No desenvolvimento da segunda estratégia, uma importante questão foi definir o ponto de sincronia, ou seja, em que ponto do trecho todas as *threads* da GPU devem esperar a execução das demais. Visando a confiabilidade dos resultados simulados, o ponto de sincronia escolhido (*syncthreads*) foi a passagem entre um tempo (Δt) para outro, antes da atualização das dos valores calculados na matrizes pressão (H), vazão (Q) e tempo (t), conforme ilustrado no trecho do código abaixo.

```
__global__ void calc_HQ_RegimeTransiente(float* HMatriz, float* QMatriz, float* TMatriz)
...

     H_tempo_atual[indice] = H_proximo_tempo[indice];
     Q_tempo_atual[indice] = Q_proximo_tempo[indice];

     __syncthreads();

     HMatriz[(controle * ns) + i] = H_tempo_atual[indice];
     QMatriz[(controle * ns) + i] = Q_tempo_atual[indice];
     TMatriz[(controle * ns) + i] = t;
...
```

Na figura 4.7, em (a) é apresentado o trecho na matriz que o bloco irá trabalhar, e destaca quais pontos serão calculados antes de uma nova sincronização. Em (b), no t0, duas *threads* calculam o segmento 1 e 2. No tempo t1, as duas *threads* calculam os segmentos 3 e 4, e em t2, o segmento 0 é calculado.

E finalmente, em (c) é apresentado o novo ponto de sincronia, ou seja, o tempo 2 será calculado apenas quando todos os segmentos no tempo 1 estiverem devidamente calculados. Esta abordagem compromete o uso do total teórico da placa, porém, garante a confiabilidade dos resultados, sanando o primeiro problema apresentado na primeira implementação.

Para sanar o segundo problema apresentado, a segunda implementação utiliza múltiplos blocos, conforme o número de segmentos fosse maior ao número de *threads* disponível no *device*. O trecho abaixo evidencia como se encontrar o número correto de blocos devem ser invocados. Neste exemplo, o código-fonte declara que o *kernel* utilizará 512 *threads* por bloco.

```
const int threadsPerBlock = 512;
...
int main(int argc, char** argv) {
     ...
```

```
    ns = n + 2; // Numero de segmentos + segmento de montante + segmento de jusante
    ...
    nthreads = threadsPerBlock;
    nblocks = (ns + threadsPerBlock ) / threadsPerBlock;
    ...

    calc_HQ_RegimeTransiente<<<nblocks, nthreads>>>(d_HMatriz, d_QMatriz, d_TMatriz);
    ...
}
```

Na figura 4.8, em (a) é apresentado o trecho na matriz em que o bloco 1 irá atuar. Realizado a computação, com uso da sincronização, em (b) é apresentado o trecho selecionado, com os segmentos selecionados em todos os tempos já calculado. Em (c), um segundo bloco seleciona um novo conjunto de dados, incluindo o último segmento do bloco anterior. Após a computação de todos os elementos no bloco 2, em (d) é apresentado os elementos selecionados pelo terceiro bloco, novamente, incluindo o último segmento do bloco anterior. Esta abordagem sanou o segundo problema apresentado, não limitando o número de segmentos ao número de *threads* presentes no *device*.

Diversos autores citam a importância do uso correto das memórias disponíveis na arquitetura CUDA [25][26][34]. Na segunda implementação, buscou-se equilibrar as memórias disponíveis as necessidades e limitações de cada variável. Para as variáveis contendo-se as matrizes Pressão (H), Vazão (Q) e Tempo (T) foi utilizado a memória global. Os vetores intermediários dentro do *kernel* utilizam a memória compartilhada (*shared memory*), pois além de ser uma memória mais rápida, comparada a memória global, permite a sincronia entre as *threads*. As variáveis que usam este tipo de memória é declarado dentro do *kernel*, conforme evidencia o código abaixo.

```
__global__ void calc_HQ_RegimeTransiente(float* HMatriz, float* QMatriz, float* TMatriz)
{
...
    __shared__ float H_tempo_atual[threadsPerBlock];
    __shared__ float Q_tempo_atual[threadsPerBlock];
    __shared__ float H_proximo_tempo[threadsPerBlock];
    __shared__ float Q_proximo_tempo[threadsPerBlock];
...
}
```

As demais variáveis são alocadas na memória constante, conforme demonstrado abaixo.

```
struct ConstantesPrograma{
     int ns;
     float qo;
     float hr;
     float r;
     float b;
     float t;
     float dt;
     float tmax;
     int controle;

     __device__ int get_ns() {return ns;}
     __device__ float get_qo() {return qo;}
     __device__ float get_hr() {return hr;}
     __device__ float get_r() {return r;}
     __device__ float get_b() {return b;}
     __device__ float get_t() {return t;}
     __device__ float get_dt() {return dt;}
     __device__ float get_tmax() {return tmax;}
     __device__ int get_controle() {return controle;}
};
...
__constant__ __device__ ConstantesPrograma ConstantDeviceContantes;
...
ConstantesPrograma *host_constantes;
host_constantes = new ConstantesPrograma;
...
host_constantes[0].ns  = ns;
host_constantes[0].hr  = hr;
host_constantes[0].r  = r;
host_constantes[0].b  = b;
host_constantes[0].dt  = dt;
host_constantes[0].tmax  = tmax;
host_constantes[0].controle  = controle;
host_constantes[0].t  = t;
host_constantes[0].qo  = qo;

...

// Alocação das constantes no DEVICE com uso de CONSTANT memory
cudaMalloc((void**)&ConstantDeviceContantes, sizeof(ConstantesPrograma));
```

```
    cudaMemcpyToSymbol(ConstantDeviceContantes, host_constantes,
sizeof(ConstantesPrograma));
}
```

Para [34], uma das formas de se otimizar o poder computacional da arquitetura é manter a placa ocupada a maior parte do tempo, evitando-se troca de dados entre *device* e *host*. Em ambas as implementações CUDA buscou esta otimização do uso do *hardware*, garantindo-se um desempenho superior ao trecho com maior restrição de computação ($CPUBound$), com foco na confiabilidade dos resultados simulados.

Como apenas a segunda implementação CUDA atingiu os dois requisitos (velocidade e confiabilidade), os ganhos de desempenho mensurados com a primeira implementação foram descartados nas análises realizadas no capítulo 5.

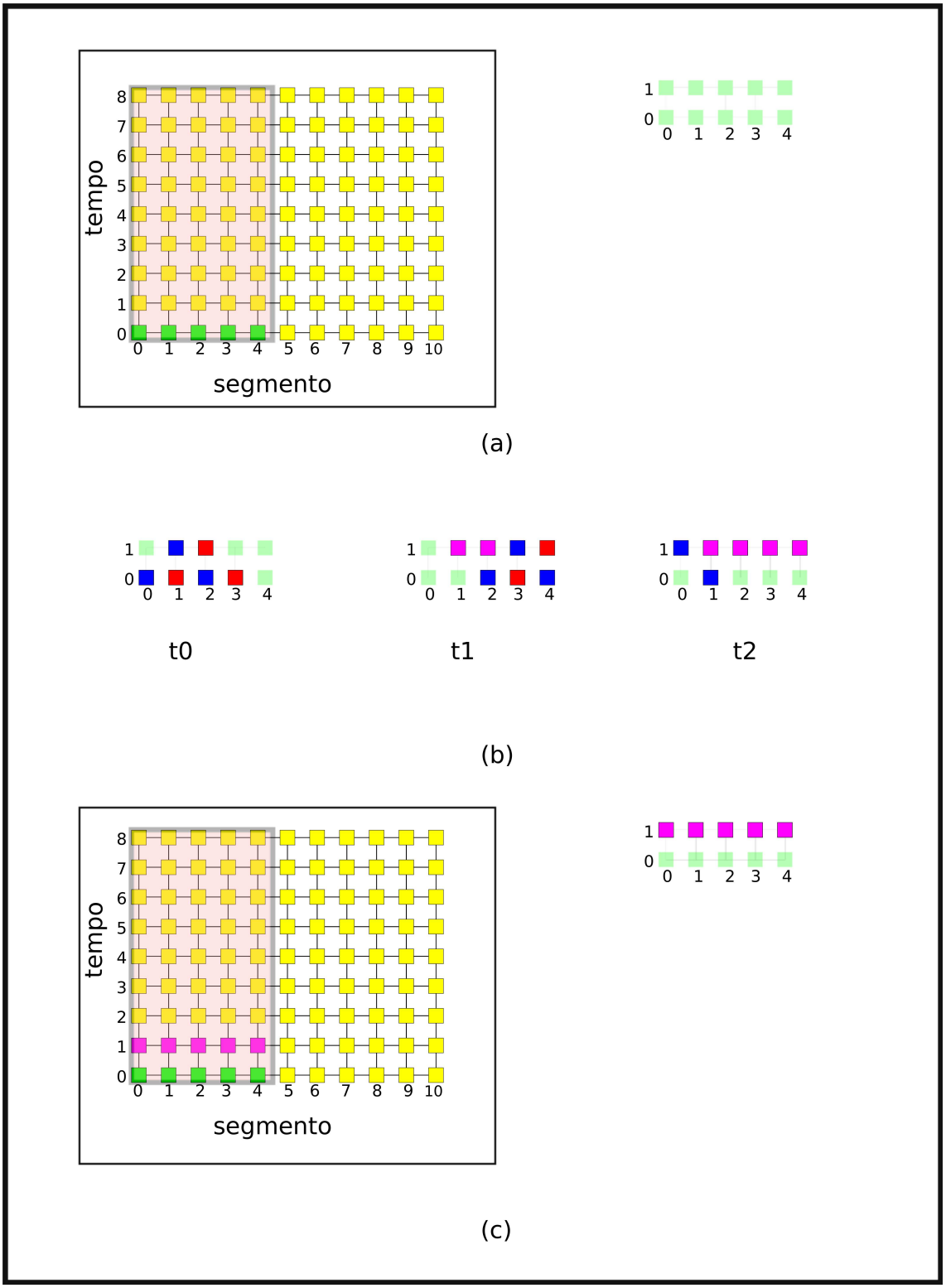

Fig. 4.7: Segunda implementação CUDA: com ponto de sincronização. Em (a), em destaque, o bloco e na direita os segmentos que serão trabalhados até a próxima sincronização. Em (b),no tempo t0, duas threads processando os segmentos 1 e 2 no tempo 1. Em t1, duas threads calculam os segmentos 3 e 4 no tempo 1, e em t2, o segmento 1 no tempo 1 é calculado. Em (c), o bloco destacado em (a), com destaque para os segmentos calculados.Os segmentos calculados são marcados em (b) e (c) com a cor rosa.

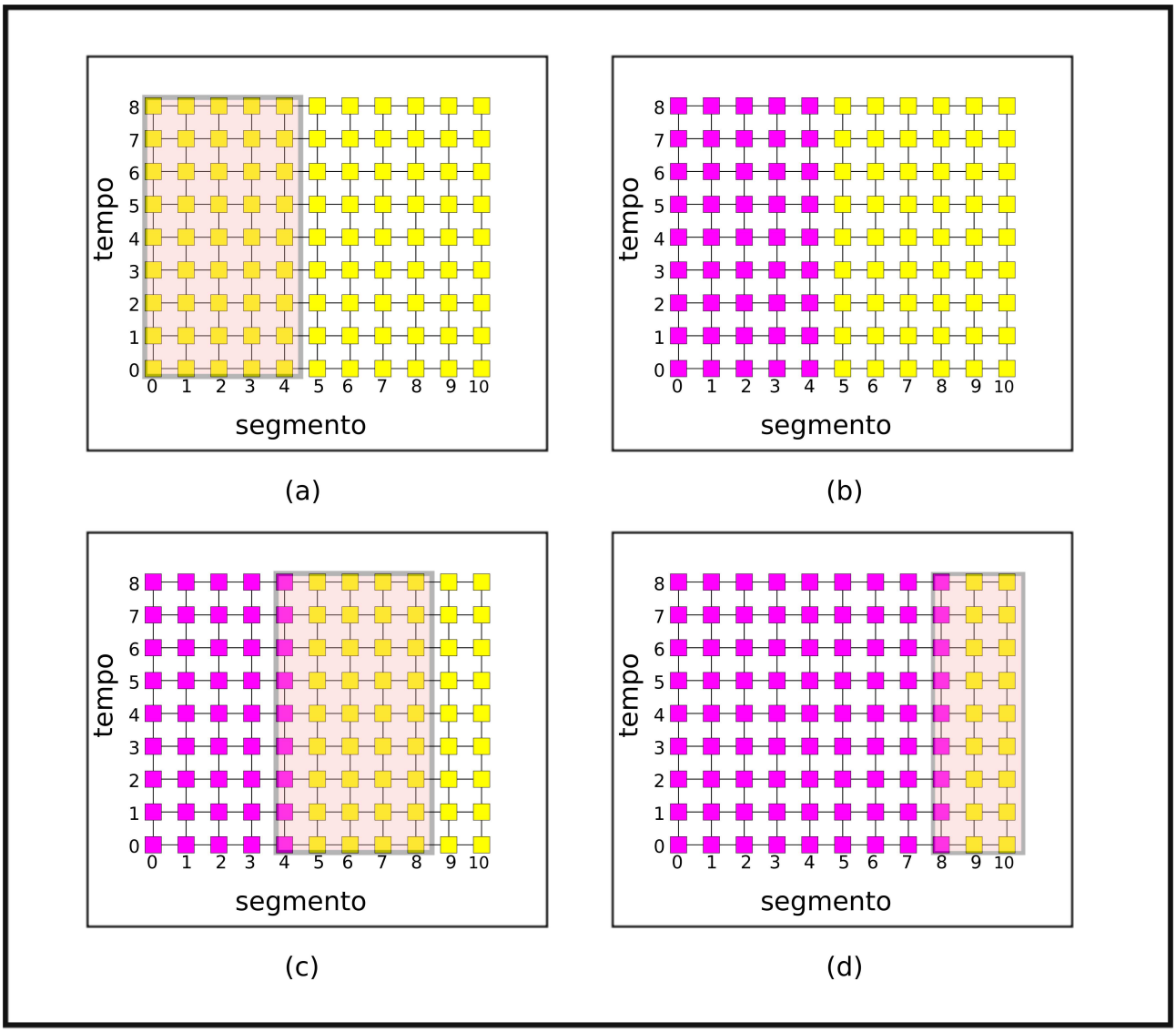

Fig. 4.8: Segunda implementação CUDA: visão global dos blocos. Em (a), a seleção do bloco 1. Em (b) os segmentos calculados no bloco 1. Em (c), a seleção do bloco 2. Em (d), os segmentos calculados e a seleção do bloco 3. Os segmentos calculados são marcados em (b), (c) e (d) com a cor rosa.

Capítulo 5

Simulações e Resultados

Este capítudo se dedica em apresentar como os testes foram conduzidos, as ferramentas desenvolvidas e os métodos utilizados para mensurar a execução dos códigos, conforme estratégias apresentadas no capítulo anterior. Esta capítulo também expõe qual a metodologia para averiguar a efetividade dos resultados obtidos de forma paralela. Por fim, apresenta os ganhos obtidos, comparando-os com resultados da implementação, afim de encontrar a eficiência e o *speedup*.

5.1 Cenários dos testes e validação

Os testes foram realizados no Cluster Beowulf, presentes no Laboratório de Simulação e de Computação de Alto Desempenho [35]. Cada estação do cluster possui um Intel Xeon Quad Core E5420 2.5 GHz com três placas Nvidias Tesla modelo C 1060. Na execução dos testes, apenas uma placa gráfica foi utilizada.

A manipulação dos parâmetros de entrada de cada versão, inicialização da aplicação, armazenamento em banco de dados e extração dos resultados se deu via um pequeno programa (*script*). A figura 5.1 ilustra o passo a passo realizado.

Definiu-se que o intervalo entre um segmento e outro seria de um metro. Assim, ao se definir um novo valor de tamanho de tubo (l), seria necessário adotar o mesmo número para o número de segmentos. Na etapa de se obter novos valores, o número de segmentos e o tamanho do tubo eram alterados. Após definidos os novos valores, o *script* sobrescreve o arquivo de entrada (arquivo .dat) com os novos valores. Dando sequencia, a aplicação executa a simulação do transiente hidráulico. A simulação imprime na tela o tempo total e o parcial da aplicação. O tempo parcial trata-se do passo 5,

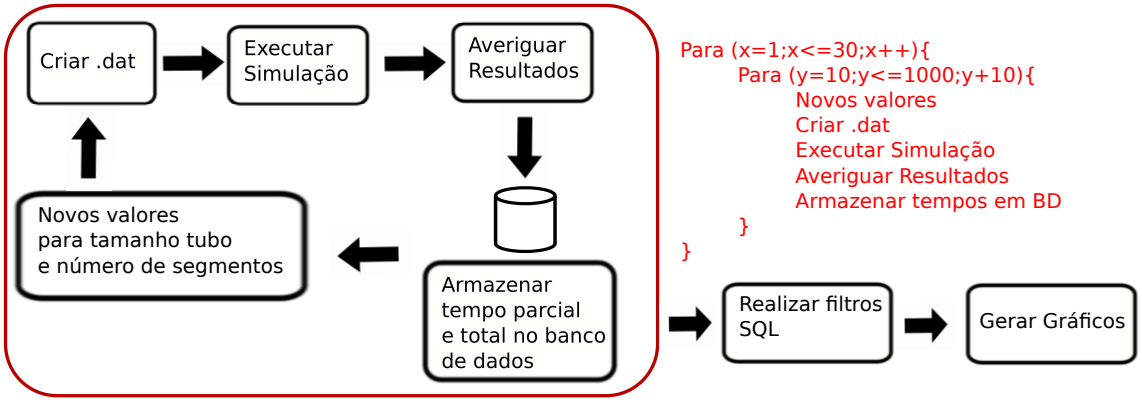

Fig. 5.1: Fluxograma com os passos realizados nos testes.

representado na figura 4.2. Estes valores são capturados para serem armazenados no banco de dados.

Uma das preocupações no desenvolvimento de aplicações paralelas é a confiabilidade nos resultados obtidos.

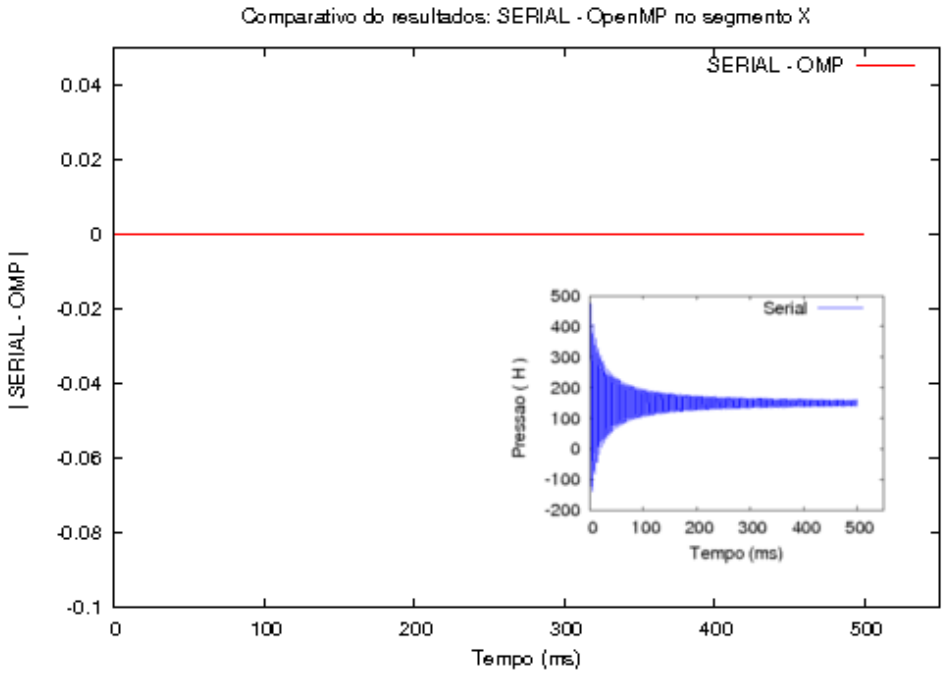

Fig. 5.2: Comparativo dos resultados serial e paralelo via OpenMP em um determinado segmento. Em destaque o módulo da subtração dos resultados obtidos serialmente com os resultados obtidos de forma paralela. No canto inferior direito, uma visualização dos resultados obtidos de forma serial.

Para verificar a confiabilidade, após o fim da simulação do fenômeno transiente hidráulico, o *script* executa uma aplicação externa, responsável em comparar os valores obtidos de forma paralela com os valores obtidos de forma serial. O código desta aplicação encontra-se no apêndice A.4.

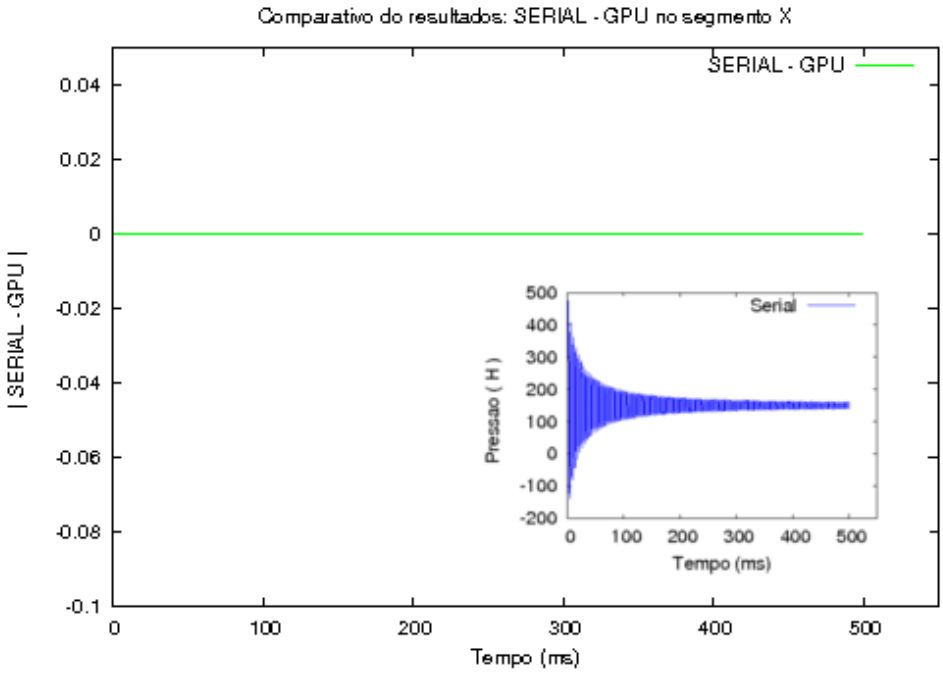

Fig. 5.3: Comparativo dos resultados serial e paralelo via GPU em um determinado segmento. Em destaque o módulo da subtração dos resultados obtidos serialmente com os resultados obtidos de forma paralela. No canto inferior direito, uma visualização dos resultados obtidos de forma serial.

Além da averiguação numérica, na figura 5.2 pode-se observar a diferença dos resultados seriais pelos resultados com uso do OpenMP. A diferença dos resultados seriais pelos resultados com uso da arquitetura CUDA podem ser observados na figura 5.3. Em ambas as imagens, as diferenças são iguais a zero, confirmando a confiabilidade dos resultados obtidos.

Averiguado a confiabilidade dos resultados obtidos, os tempos parciais e totais, obtidos na etapa que executou a simulação são armazenados no banco de dados, juntamente com o tipo de simulação (serial, OpenMP ou GPU) com a respectiva variação e o número de segmentos. Após a etapa de armazenamento no banco de dados, retorna-se na etapa de descobrir um novo valor, somando-se dez aos valores simulados anteriormente e repetindo-se o processo. Findado o ciclo de simulações, retoma-se o número de tubo novamente em dez, repetindo-se este ciclo de simulação por trinta vezes, afim de se obter uma confiabilidade estatística nos resultados extraídos.

Os testes, simulando-se trinta ciclos, foram realizados com a versão serial, com as versões OpenMP e GPU. Na versão OpenMP, os testes variaram em 4, 8, 16, 32, 64 e 128 *threads* disponíveis. Com a versão GPU, os testes variaram em 16, 32, 64, 128, 256 e 512 *threads* disponíveis a cada bloco. Findado todos os ciclos de simulações, o *script*, via linguagem de consulta estruturada (*SQL*) consegue extrair do banco de dados as informações para as análises feitas a seguir.

5.2 Análises

De acordo com o pseudo-código apresentado em 4.2, o trecho escolhido da aplicação para ser paralelizado foi o passo 5. Nos testes, o tempo total - a execução de todas as etapas apresentadas no pseudo-código - e o tempo parcial - tempo da execução apenas do passo 5 - foram extraídos. Na figura 5.4, a média dos dois tempos são apresentadas, evidenciando-se que o passo 5 é computacionalmente o trecho mais caro. Pela proximidade entre os dois tempos, a figura 5.5 enfatiza os primeiros resultados apresentados na figura 5.4.

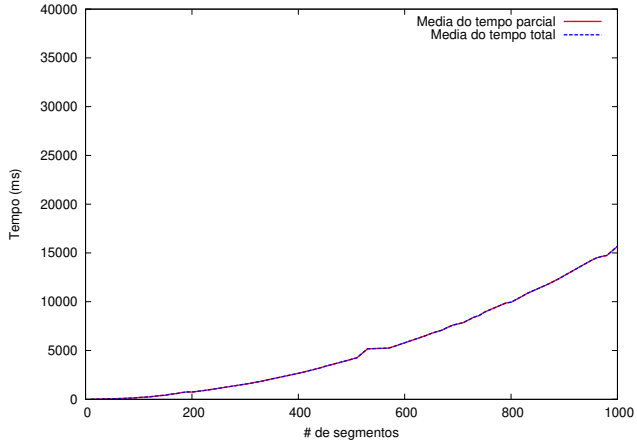

Fig. 5.4: Tempo Serial: Total x Parcial

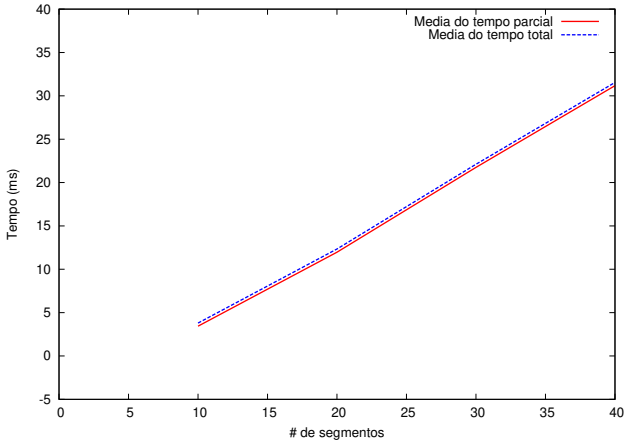

Fig. 5.5: Tempo Serial: Total x Parcial (início)

Assim, a escolha por paralelizar apenas o passo 5 mostrou-se correta, pois se trata do trecho de maior consumo de tempo em todos os segmentos escolhidos. Os resultados a seguir apresentam as estratégias adotadas para redução no tempo de execução deste passo.

Os testes feitos com a *API* OpenMP apresentaram queda no tempo médio de execução a medida que novas *threads* são inseridas.

A média de tempo com uso de 4, 8 e 16 *threads* apresentam variações entre um segmento e outro, em especial a partir de 600 segmentos (fig. 5.6). Para Tanembaum [28] isto ocorre porque "com a alternância da CPU entre os processos, a taxa que o processo realiza sua computação não é uniforme e provavelmente não reproduzível". Conforme o número de segmentos cresce, o tempo de execução necessário para a realização de toda a computação aumenta. Por isto, variações entre um segmento e outro ocorrem, a medida que o sistema simulado aumente.

A média de tempo com uso de 32, 64 e 128 *threads* não apresentam a mesma variação. Isto acontece, segundo [18], porque nas arquiteturas baseadas na arquitetura de *Von Neumman*, tanto os dados quanto os programas são armazenados na memória. A unidade central de processamento é

separada da memória. As instruções e os dados devem ser canalizados (*piped*) afim de se obter um melhor ganho computacional. Assim, a medida que o número de *threads* cresce, diminui-se o desvio padrão e o tempo de execução e conforme os dados são canalizados, o conjunto de processadores tendem a atingir o limite teórico, onde todos os processadores estão ocupado o tempo todo. Na figura 5.6, todas as médias encontradas podem ser observadas, ficando explícito uma suavização da linha pode ser observada nas médias a partir de 32 *threads*. Nesta figura também, fica explícito que os tempos diminuem entre o uso de 64 e 128 *threads*, mas com uma diferença menor, comparado as médias entre 64 e 32 *threads*. Para comparações posteriores, serão considerados os dados com uso de 128 *threads*, visto ter apresentado os menores tempos de execução.

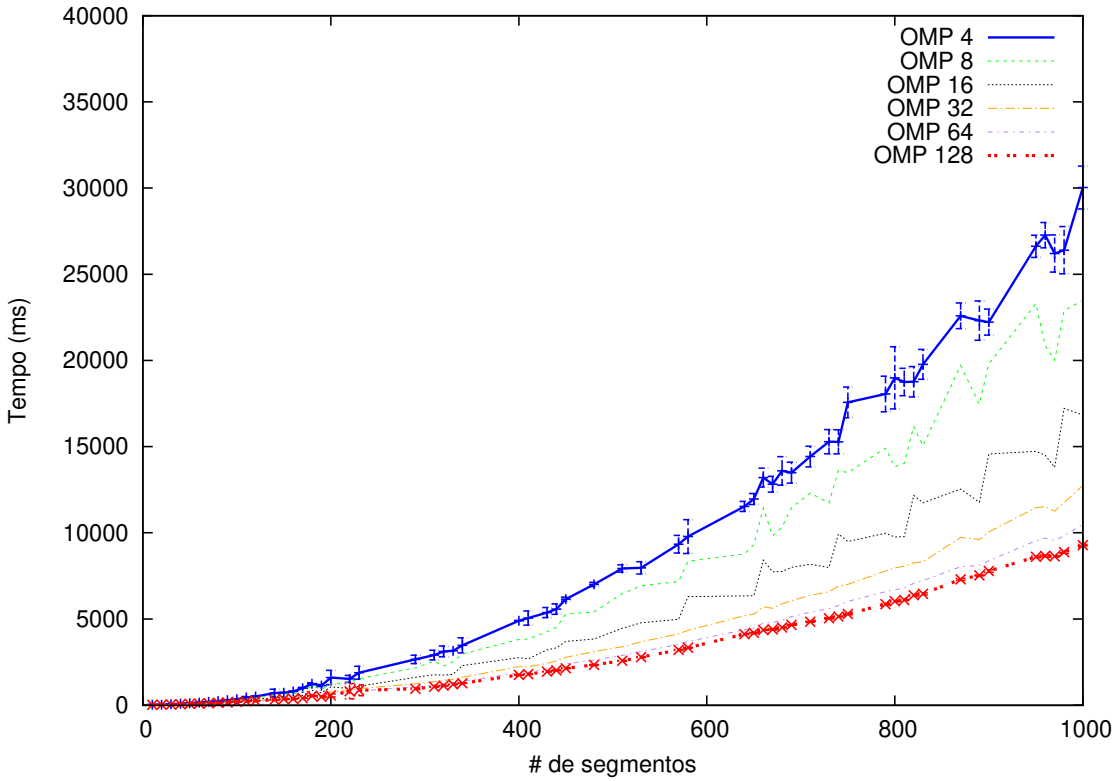

Fig. 5.6: Médias de tempo por segmentos com uso de OpenMP. O desvio padrão é apresentado nas médias com 4 e 128 threads apenas.

A primeira implementação com uso da arquitetura CUDA apresentou expressivos ganhos computacionais, mas problemas em relação aos resultados seriais foram detectados. Assim, os tempos de execução da primeira implementação foram descartados. A segunda implementação, devido a sincronia das *threads*, apresenta ganhos computacionais menores comparado a primeira implementação. Porém, por apresentar resultados corretos, foi a implementação analisada.

Os tempos de execução na implementação GPU apresentam pouca variação com as mesmas condições, ou seja, a maioria dos segmentos ficaram com desvio-padrão entre zero e um milisegundo. Isto ocorre porque os recursos computacionais da placa (*device*) estão dedicados a aplicação simulada, não havendo concorrência com outros processos em execução no *host*. A figura 5.7 apresenta um comparativo entre o tempo (em milisegundos) médio de execução pelo número de segmentos. Os ganhos computacionais crescente, a medida que o número de *threads* por bloco diminuem, deve-se ao menor esforço em sincronizar um número menor de *threads* a cada variação de Δt.

Os resultados extraídos, apresentados nesta seção apontam o uso de 16 *threads* por bloco a que apresenta melhores ganhos computacionais e confiabilidade nos resultados simulados.

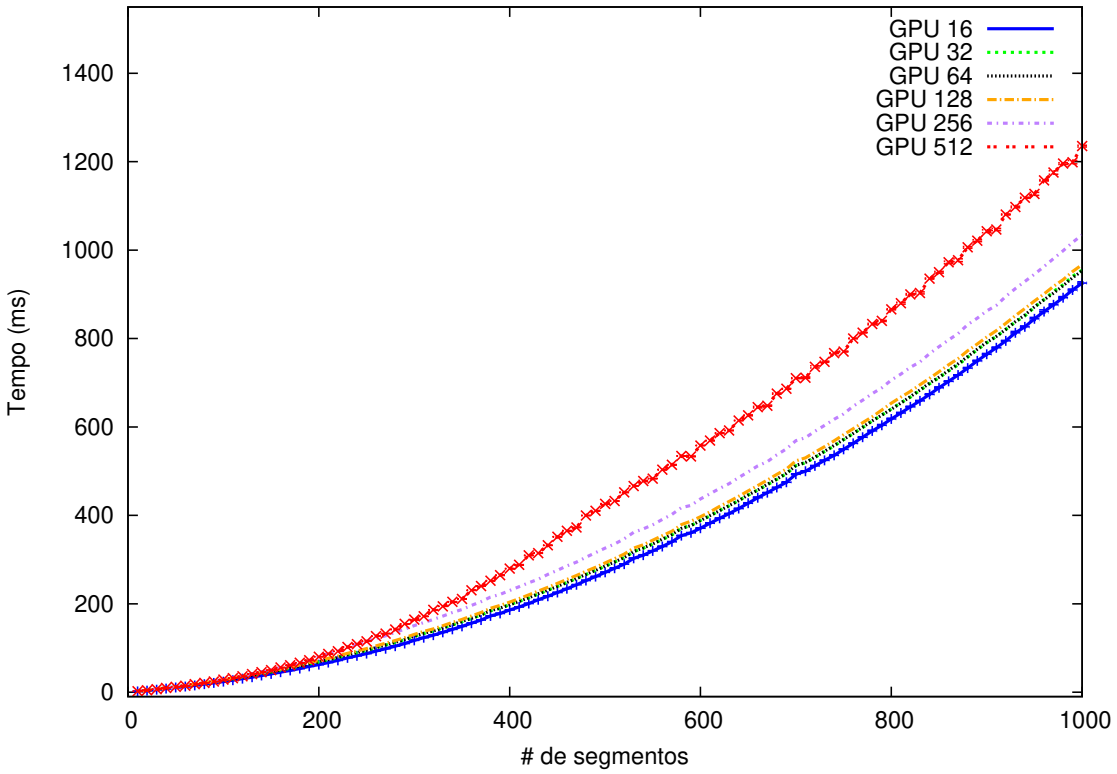

Fig. 5.7: Médias de tempo por segmentos com uso da arquitetura CUDA.

A figura 5.8 apresenta a média dos tempos serial, a média dos tempos OpenMP com uso de 128 *threads* e a média dos tempos GPU com uso de 16 *threads* por bloco em cada segmento. Pode-se visualizar que o tempo de execução da GPU, comparado ao outro método paralelo e ao método serial é expressivamente menor.

Foster [32] recomenda ter-se uma métrica independente do algoritmo que não seja tempo de execução. Embora o tempo de execução seja a métrica mais conveniente para se avaliar o desempenho

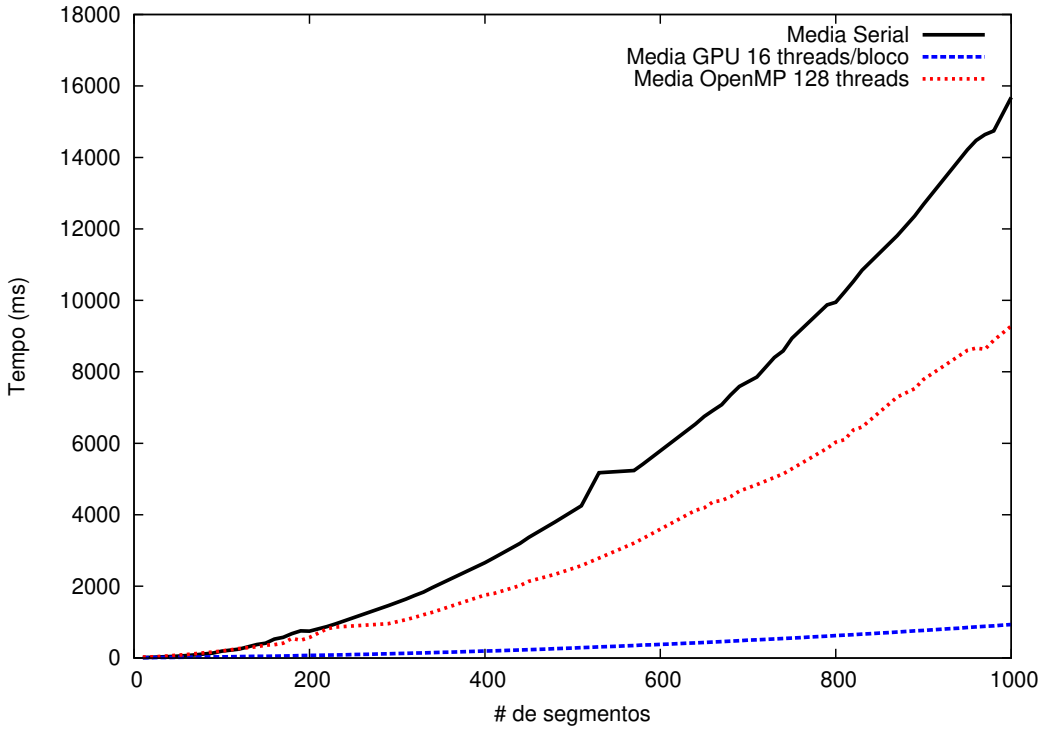

Fig. 5.8: Médias de tempo por segmentos: Serial x OpenMP x GPU .

do algoritmos paralelos discutidos, foi realizado uma análise sobre a eficiência de cada algoritmo.

Por eficiência, entende-se ser a fração de tempo que processadores passam realizando trabalho útil. Por ser uma métrica relacionada a eficácia com que um algoritmo usa os recursos computacionais de um computador paralelo, torna-se uma métrica independente do tamanho do problema.

Para o cálculo da eficiência e do *speedup*, considerou-se o menor tempo em cada número de segmento para cada técnica sobre o menor tempo serial vezes o número de processadores disponíveis. Para o OpenMP, considerou-se os 4 processadores disponíveis em CPU, enquanto para o GPU, considerou-se as 32 *threads*, disponível em cada *warp*. Conforme apresentado na sessão 3.3, a eficiência varia entre 0 e 1, sendo 1 quando todos os processadores são utilizados o tempo todo.

Embora a eficiência do OpenMP mostrou-se melhor, comparado a GPU (figura 5.9), pode-se observar na figura 5.10, conforme o número de segmentos cresce, o *speedup* com uso da GPU (em azul) mantém-se em crescimento até próximo de 16x, com pico de 17x, enquanto o *speedup* com OpenMP estagna-se próximo a 3.3x. Caso a eficiência da implementação GPU crescesse, com o uso de uma única placa o *speedup* teria um limite de 32x, limite de *threads* disponível por *warp*.

Os ganhos apresentados aqui permitem que o método das características, utilizado neste trabalho para resolução de problemas hidráulicos são problemas computacionalmente interessantes para serem

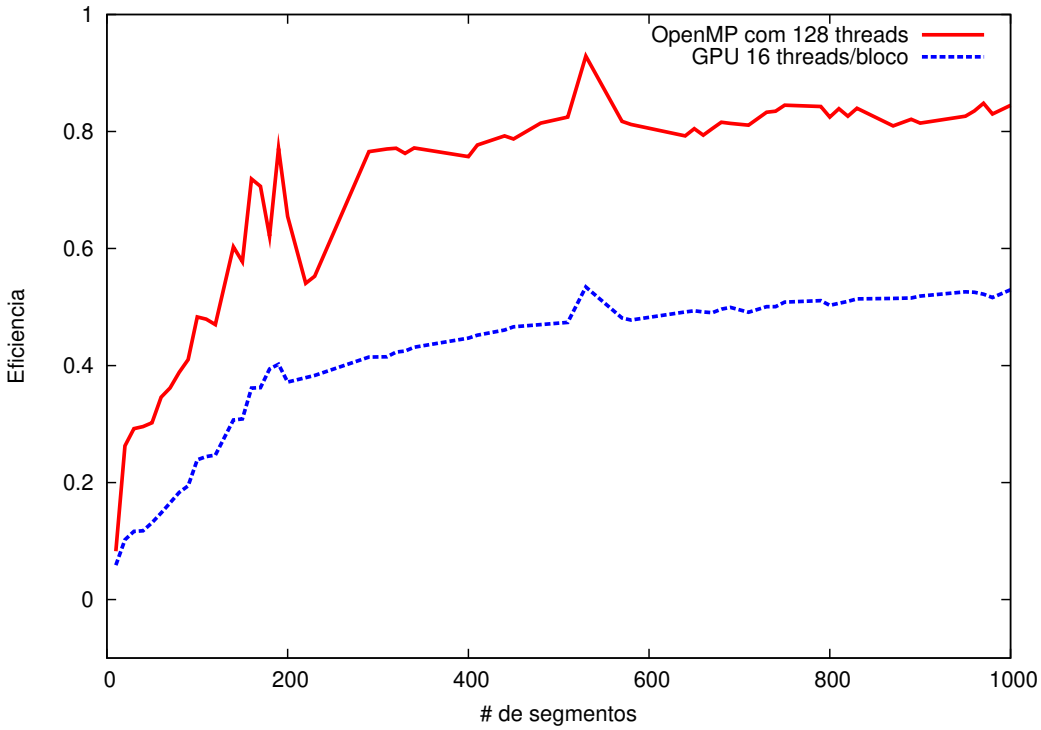

Fig. 5.9: Comparação da eficiência computacional OpenMP com 128 *threads* x GPU com 16 *threads* por bloco

resolvidos com o uso da computação paralela, em especial, com uso de arquiteturas SIMD.

Os ganhos mostrados na figura 5.10 mostram tendência de crescimento, e poderiam ser maiores, conforme o sistema fosse maior. Para sistemas com um número maior de tubos, conexões e equipamentos hidromecânicos, recomendamos a criação de um mapeamento que relacione um determinado tubo do sistema as posições no vetor. Na figura 5.11 é apresentado um modelo desse mapeamento, onde cada tubo é representado por um número de elementos pré-determinados, tendo o nó de montante e nó de jusante o ponto de conexão entre eles.

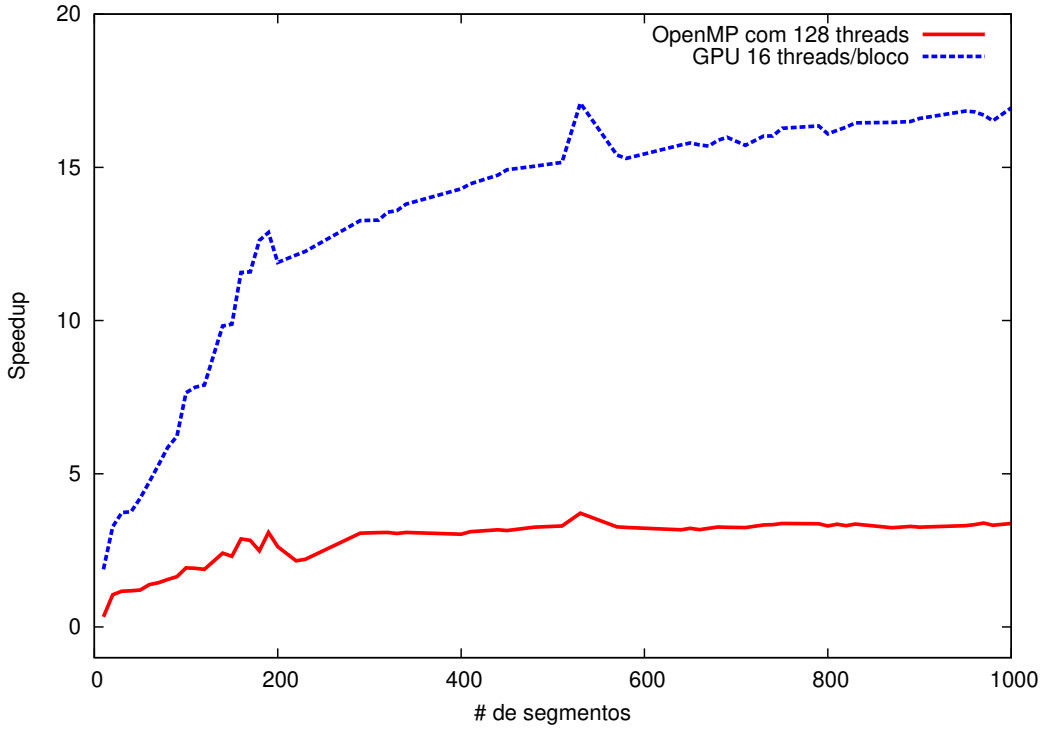

Fig. 5.10: *Speedup*: OpenMP com 128 *threads* x GPU com 16 *threads* por bloco

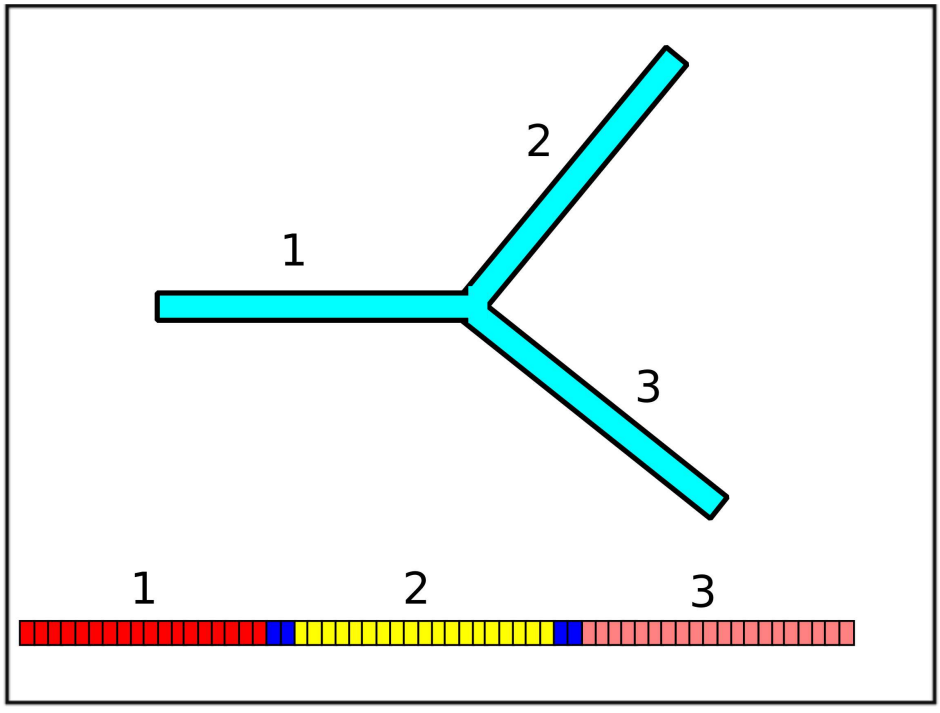

Fig. 5.11: Exemplo de mapeamento para sistemas maiores e o vetor representando o sistema como um todo. Em vermelho, os elementos representam o tubo 1, em amarelo o tubo 2 e em rosa o tubo 3. Os elementos em azul representam os nós de montante e jusante.

Capítulo 6

Conclusões

Os resultados obtidos evidenciaram como estratégias de computação paralela podem ser aplicadas ao método das características, um método altamente paralelizável, e levar ganhos significativos de desempenho em problemas envolvendo transientes hidráulicos.

Os ganhos obtidos com o uso da técnica OpenMP apresentaram *speedups* de até $3.3\times$ e eficiência próxima de seu limite, enquanto o uso da computação paralela com CUDA apresentou ganhos próximos de $17\times$. A estratégia envolvendo CUDA que foi adotada nesse trabalho dividiu o tubo em várias regiões onde, para cada uma delas, um bloco de *threads* era utilizado para efetuar os cálculos. Um sincronismo das *threads* foi imposto a cada incremento de tempo para garantir o cálculo correto dos valores de pressão e vazão.

O tempo necessário para o aprendizado do OpenMP mostrou-se menor comparado ao tempo necessário para o aprendizado da linguagem CUDA-C. Isto ocorre porque a *API* OpenMP encapsula a complexidade da programação paralela. No entanto, apesar da curva de aprendizado de OpenMP ter sido menor e a implementação CUDA-C ter apresentado eficiência computacional menor, as simulações com uso da arquitetura CUDA apresentaram resultados com ganhos computacionais expressivamente melhores. Isto justifica o uso da linguagem CUDA-C e de placas gráficas em simulações de transientes hidráulicos com o método das características.

Como o método das características é aplicado em outras áreas da Ciência, as implementações desenvolvidas neste trabalho podem ser adaptadas para outros tipos de sistemas também a fim de obter ganhos de desempenho computacional.

Capítulo 7

Perspectivas futuras

Um dos pontos de melhoria a ser explorado é o uso de multi-GPUs, ou seja, o uso intenso de todas as placas gráficas disponíveis localmente e em outras estações. Para isto, se faz necessário uma nova decomposição do problema para mesclar a técnica GPU com outras técnicas de computação paralela como *Message Passing Interface* (MPI) [20].

Novos trabalhos poderão também ser realizados afim de utilizar as implementações desenvolvidas aqui em problemas maiores (como num sistema hidráulico que represente um bairro ou uma cidade, por exemplo) e em simuladores comerciais. Para isto, além de um módulo de leitura de uma rede com tubulações e outros dispositivos hidromecânicos, se faz necessário um desenvolvimento com uso de boas práticas de engenharia de software e do envolvimento de outros profissionais ligados a área de hidráulica.

Os ganhos obtidos com os códigos desenvolvidos possibilitam também acoplar o cálculo de transiente hidráulico com técnicas de computação bioinspirada como algoritmos genéticos. Isso poderia ser útil em problemas de otimização de sistemas hidráulicos onde, por exemplo, a velocidade de uma bomba deveria ser otimizada para fornecer a maior disponibilidade de serviço para um conjunto de usuários [12]. Para essa otimização, o sistema hidráulico estudado deverá ser simulado inúmeras vezes, variando-se as manobras hidromecânicas com foco em encontrar as menores ondas de sobrepressão e subpressão.

Referências Bibliográficas

[1] STREETER, V. L. e WYLIE, E. B. *Fluid Transients*. McGrawHill, 1978.

[2] IZQUIERDO J. e IGLESIAS P. L. Mathematical Modelling of Hydraulic Transients in Simple Systems. *Mathematical and Computer Modelling*, pages 801–812, 2002.

[3] WOOD D. J. , LINGIREDDY, S. , BOULOS P. , KARNEY B.W. e McPHERSON D. L. Numerical methods for modeling transient flow in distribution systems. *Journal of the American Water*, pages 104–115, 2005.

[4] ABNT/CB-02 , editor. *NBR12215 - Projeto de adutora de água para abastecimento público - Procedimento*, volume I. ABNT, 1992.

[5] Intel Halts Development Of 2 New Microprocessors), Acesso em 10-ago-2012. Disponível em: `http://www.nytimes.com/2004/05/08/business/intel-halts-development-of-2-new-microprocessors.html`.

[6] General-Purpose Computation on Graphics Hardware, Acesso em 15-jul-2011. Disponível em: `http://gpgpu.org/`.

[7] VERDIÈRE G. C. Introduction to GPGPU, a hardware and software background. *Académie des sciences*, 2010.

[8] OWNERS J. D., HOUSTON M., LUEBKE D., GREEN S., STONE J. E. e PHILLIPS J. C. GPUComputing - Graphics Processing Units - powerful, programmable, and highly parallel - are increasingly targeting general-purpose computing applications. *Proceedings of the IEEE*, pages 879–899, 2008.

[9] OpenCL - The open standard for parallel programming of heterogeneous systems, Acesso em 20-jun-2011. Disponível em: `http://www.khronos.org/opencl/`.

[10] WU Z. Y. e EFTEKHARIAN A. A. Parallel artificial neural network using cuda-enabled gpu for extracting hydraulic domain knowledge of large water distribution systems. volume 414, pages 9–9. ASCE, 2011.

[11] CROUS P. A. e Van ZYL J. E. e NEL A. Using stream processing to improve the speed of hydraulic network solvers. volume 340, pages 71–71. ASCE, 2008.

[12] RIBEIRO L. C. L. J. Modelo hibrido multiobjetivo para obtenção de roteiros operacionais de bombas de rotação variavel em instalações hidraulicas. *Tese de doutorado*, 2007.

[13] LUVIZOTTO Jr E. Controle Operacional de Redes de Abastecimento de Água auxilido por computador. *Tese de doutorado*, page 143p, 1995.

[14] LUVIZOTTO JR, E. ; ANDRADE, J. G. P. *Controle e gestão operacional de sistemas de abastecimento de água*. São Carlos, SP, 2004.

[15] OpenMP API specification for parallel programming, Acesso em 1-jun-2011. Disponível em: `http://openmp.org/wp/`.

[16] CHAUDHRY, M. H. *Applied Hydraulic Transients*. Van Nostrand Reinhold Company, 1987.

[17] ALMASI G. S. e GOTTLIEB A. *Highly parallel computing*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989.

[18] TANEMBAUM, A. S. *Organização Estruturada de Computadores*. Pearson / Prentice Hall, 2006.

[19] FLYNN M. J. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, pages 948 –960, 1972.

[20] The Message Passing Interface (MPI) standard, Acesso em 10-jun-2011. Disponível em: `http://www.mcs.anl.gov/research/projects/mpi/`.

[21] WRIGHT N. e VILLANUEVA I. Modeling urban flood inundation in a parallel computing environment. Number 40976, pages 462–462. ASCE, 2008.

[22] HSU M. H. TENG W. H., CHEN S. H. Parallel computing for surface inundation with local grid refinement. ASCE, 2001.

[23] AGRAWAL J. e MATHEW T. V. Transit route network design using parallel genetic algorithm. *Journal of Computing in Civil Engineering*, pages 248–256, 2004.

[24] GORING D.G. e WALTER R.A. A scalable, parallel, wave equation model for storm surge propagation. ASCE, 2001.

[25] NVIDIA Corporation. *NVIDIA CUDA Programming C Guide*. Santa Clara, CA, USA, 2011.

[26] SANDERS J., KANDROT E. *CUDA by example : an introduction to general-purpose GPU programming*. Boston, MA, USA, 2011.

[27] Parallel Processing With CUDA, Acesso em 15-out-2011. Disponível em: `http://www.mpronline.com/mpr/h/2008/0128/220401.html`.

[28] TANEMBAUM, A. S. *Sistemas Operacionais Modernos* . Pearson, São Paulo, SP, 2007.

[29] IKEDA P. A. Um estudo do uso eficiente de programas em placas gráficas. *Dissertação de Mestrado*, 2011.

[30] KAN S.H. *Metrics and Models in Software Quality Engineering*. Adison Wesley, 2002.

[31] JAIN R. The art of Computer Systems Performance Analysis. 1991.

[32] FOSTER, I. *Designing and Building Parallel Programs*. Minneapolis, MN, USA, 1995. Disponível em: `http://www.mcs.anl.gov/~itf/dbpp/text/book.html`.

[33] GRAMA,A.;GUPTA,A.;KARYPIS,G.;KUMAR,V. *Introduction to Parallel Computing*. Minneapolis, MN, USA, 2003. Disponível em: `http://www-users.cs.umn.edu/~karypis/parbook/`.

[34] Stanford Courses - CS193G - Programming Massively Parallel Processors with CUDA, Acesso em 10-out-2010. Disponível em: `stanford-cs193g-sp2010.googlecode.com`.

[35] Laboratório de Simulação e de Computação de Alto Desempenho - LaSCADo, Acesso em 10-nov-2012. Disponível em: `http://www.ft.unicamp.br/vitor/lascado/`.

Apêndice A

Código Fonte

A.1 versão C serial

```
/*************************************************************
*
*
* Programa para teste de perfomance CPU serial
* do Metodo das caracteristicas.
*
*
*  Orlando Saraiva Jr (saraiva(at)rc.unesp.br )
*  Prof. Vitor Rafael Coluci vitor(at)ft.unicamp.br
*
*        setembro/2011
*
*************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>


#define PI 3.14159265
#define g 9.806

float hr;       // Virá do arquivo .dat
float f;        // Virá do arquivo .dat
float l;        // Tamanho horizontal do pipeline em metros. Virá no .dat
float d;        // Diametro. Virá do arquivo .dat
float dt;       // Delta T. Virá do arquivo .dat
float tmax;     // Tempo máximo da simulação em segundos. Virá do arquivo .dat
```

```
float cdao;     // Virá do arquivo .dat

float r;
float a;        // Celeridade da onda
float qo;
float b;
float cm;
float cp;
float dx;       // Delta X. Comprimento do tudo pelo número de segmentos.
float tf;

float as;
float t;
float bp;
float bm;

float* HMatriz; // Matriz dinâmica para H ( Pressao )
float* QMatriz; // Matriz dinâmica para Q ( Vazao )
float* TMatriz; // Matriz dinâmica para T ( Tempo )

float* d_HMatriz; // Matriz dinâmica para H ( Pressao ) em device
float* d_QMatriz; // Matriz dinâmica para Q ( Vazao ) em device
float* d_TMatriz; // Matriz dinâmica para T ( Tempo ) em device

float* H; // Vetor dinâmico para H ( Pressao )
float* Q; // Vetor dinâmico para Q ( Vazao )
float* hp; // Vetor dinâmico para hp, para auxílio no MOC.
float* qp; // Vetor dinâmico para qp, para auxílio no MOC.


float* d_H; // Matriz dinâmica para H ( Pressao ) que será alocada no device
float* d_Q; // Matriz dinâmica para Q ( Vazao ) que será alocada no device
float* d_hp; // Vetor dinâmico para hp, para auxílio no MOC que será alocada no device.
float* d_qp; // Vetor dinâmico para qp, para auxílio no MOC que será alocada no device.

int i;          // Variável de controle nos laços for
int linha;      // Variável de controle nos laços for
int controle;   // Controle auxiliará para linkar o vetor auxiliar H e Q as respectivas matrizes.
int n; // Numeros de segmentos. Virá do arquivo .dat
int ns;         // Numero de segmentos + segmento de montante + segmento de jusante

int N; // Numeros de segmentos * Delta T

FILE *nomearqe; // Nome do Arquivo de Entrada
FILE *nomearqs; // Nome do Arquivo de Saída
int gravarDisco = 0; // variavel de controle de saida de gravacao em disco das matrizes.

char str[80]; // Precisei desta variável para pegar dados do .dat do arquivo de entrada.

// Calcula o MOC em CPU
void calc_HQ_CPU();
```

```
// Inicializar elementos do vetor com zeros.
void testAlocation(float*);

// Inicializar elementos do vetor com zeros.
void initZero(float*, int);

/************************************************************
*
*           Função main no host
*
************************************************************/
int main(int argc, char** argv)
{
float tcpu, tgpu;
     struct timeval p1start, p1stop, totalt;
unsigned int timer;  //Variavel para contagem de tempo.

if(argc==2) {
gravarDisco = atoi(argv[1]);
// printf("\nAtivado modo impressão em arquivo.\n\n");
}

/*=========================Leitura dos dados==========================*/
nomearqe = fopen ("caso5.dat" , "r");
    if (nomearqe == NULL) perror ("Erro ao abrir arquivo");
    else
    {
fscanf (nomearqe, "%s", str);
      fscanf (nomearqe, "%f", &hr);
                fscanf (nomearqe, "%s", str);
      fscanf (nomearqe, "%f", &f);
                fscanf (nomearqe, "%s", str);
      fscanf (nomearqe, "%f", &l);
                fscanf (nomearqe, "%s", str);
      fscanf (nomearqe, "%f", &d);
                fscanf (nomearqe, "%s", str);
      fscanf (nomearqe, "%d", &n);
                fscanf (nomearqe, "%s", str);
      fscanf (nomearqe, "%f", &tmax);
                fscanf (nomearqe, "%s", str);
      fscanf (nomearqe, "%f", &a);
                fscanf (nomearqe, "%s", str);
      fscanf (nomearqe, "%f", &cdao);
      fclose (nomearqe);
}

// Limpa contadores de tempo da CPU

timerclear(&p1stop);
timerclear(&p1start);
```

```
// Calculo na CPU (serial)
gettimeofday(&p1start,NULL);

calc_HQ_CPU();

gettimeofday(&p1stop, NULL);

// Obtem tempo de gasto na CPU em ms
timersub(&p1stop,&p1start,&totalt);
tcpu = 0.001*(totalt.tv_sec*1000000 + totalt.tv_usec);

printf("Tempo de execucao SERIAL =  %f ms\n",tcpu);

}

/************************************************************
*
*    Função que calcula o MOC no host e de forma SERIAL
*
************************************************************/
void calc_HQ_CPU() {


if(gravarDisco == 1){
nomearqs = fopen ("saida_vetor_MOC_serial.dat" , "w");
}

/*======================== Calculo das Constantes ======================*/
as = (PI*d*d)/4;
dx = l/n;
dt = dx/a;
ns = n + 2; // Numero de segmentos + segmento de montante + segmento de jusante
r = f * dx/(2*g*d*as*as);
b = a/(g*as);

/*========== Alocação dos Vetores e Matrizes H e Q ===============*/

linha = ((int)(tmax/dt));
N = ns + (ns * linha); // Delta T para 0 + conjunto de delta T + Última interação Delta T
/* printf("Numero de elementos da matriz =  %d \n",N);
printf ("linha = %d\n",linha);
printf ("tmax = %4.2f\n",tmax);
printf ("dt = %4.2f\n",dt);
printf ("n = %d\n",n);
printf ("ns = %d\n",ns);
printf ("N = %d\n",N);
*/
// Controle auxiliará para linkar o vetor auxiliar H e Q as respectivas matrizes.
controle = 0;
```

```
     size_t sizeMATRIX = N * sizeof(float);
     size_t sizeVECTOR = ns * sizeof(float);

HMatriz = (float*)malloc(sizeMATRIX);
QMatriz = (float*)malloc(sizeMATRIX);
TMatriz = (float*)malloc(sizeMATRIX);

H = (float*)malloc(sizeVECTOR);
Q = (float*)malloc(sizeVECTOR);
hp = (float*)malloc(sizeVECTOR);
qp = (float*)malloc(sizeVECTOR);

testAlocation(HMatriz);
testAlocation(QMatriz);
testAlocation(TMatriz);

testAlocation(H);
testAlocation(Q);
testAlocation(hp);
testAlocation(qp);

/*================= Inicialização de H e Q ====================*/

initZero(HMatriz, N);
initZero(QMatriz, N);
initZero(TMatriz, N);

initZero(H, ns);
initZero(Q, ns);
initZero(hp, ns);
initZero(qp, ns);
        t = 0;
/*================ Calculo do Regime Permanente Inicial ==================*/

     qo=sqrt(((cdao*cdao)*2*hr*hr)/(1+(cdao*cdao)*2*g*n*r));

     for(i=0; i<= ns ; i++) {
          H[i] = hr-(i)*r*qo*qo;
          Q[i] = qo;
          HMatriz[i] = H[i];
          QMatriz[i] = Q[i];
          TMatriz[i] = t;
     }

/* ================================ Inicio do transitorio ================================ */
while(t < tmax) {
t=t+dt;
controle++;

     /* ================ Pontos Interiores ================================== */
for(i=1 ; i<ns ; i++){
```

```
        cp = H[i-1]+b*Q[i-1];
        bp = b+r*fabs(Q[i-1]);
        cm = H[i+1]-b*Q[i+1];
        bm = b+r*fabs(Q[i+1]);
        qp[i] = (cp-cm)/(bp+bm);
               hp[i] = cp-bp*qp[i];
}

     /* ============ Condicao de Contorno de montante: Reservatorio ============ */
hp[0] = hr;
cm = H[1]-b*Q[1];
bm = b+r*fabs(Q[1]);
qp[0] = (hr-cm)/bm;

     /* ============ Condicao de contorno de jusante: Válvula ================== */
cp = H[ns-2]+b*Q[ns-2];
qp[ns-1] = 0;
hp[ns-1] = cp;

    /* ============ Atualizacao das variaveis e impressao ===================== */
for(i=0 ; i<ns ; i++) {
        H[i] = hp[i];
        Q[i] = qp[i];
}
    /* ============ Atualizacao na matriz ================================== */
for(i=0 ; i<=ns ; i++) {
        HMatriz[(controle * ns ) + i] = H[i];
        QMatriz[(controle * ns ) + i] = Q[i];
               TMatriz[(controle * ns ) + i] = t;
}

      } // Fim do laço for

    /* ============ Impressao da Matriz Completa ============================ */
if(gravarDisco == 1){
for(i=0 ; i<N ; i++) {
     fprintf(nomearqs, "%4.2f      %4.5f      %4.5f\n", TMatriz[i] , HMatriz[i] , QMatriz[i]);
}
}

free(HMatriz);
free(QMatriz);
free(TMatriz);

free(H);
free(Q);
free(hp);
free(qp);

if(gravarDisco == 1){
fclose (nomearqs);
```

```
}
}

/*************************************************************
*
*  Função que inicializa matrizes com 0.0
*
*************************************************************/
void testAlocation(float* data)
{
if(data == 0 ) {
printf("Falha na alocação\n");
printf("Memoria RAM indisponivel para os parametros passados\n");
exit(-1);
}
}

/*************************************************************
*
*  Função que inicializa matrizes com 0.0
*
*************************************************************/
void initZero(float* data, int n)
{
int i;
for (i = 0; i < n; ++i){
         data[i] = 0.0;
}
}
```

A.2 versão CUDA-C

```
/*************************************************************
*
*         Programa Metodo das caracteristicas em GPU
*
*          Orlando Saraiva Jr saraiva(at)rc.unesp.br
*          Prof. Vitor Rafael Coluci vitor(at)ft.unicamp.br
*
*        julho/2012
*
*************************************************************/

// Includes
#include <cutil.h>
#include <cuda.h>
```

```
#include <cutil_inline.h>

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>


#define PI 3.14159265
#define g 9.806

float hr;       // Virá do arquivo .dat
float f;        // Virá do arquivo .dat
float l;          // Tamanho horizontal do pipeline em metros. Virá no .dat
float d;         // Diametro. Virá do arquivo .dat
float dt;       // Delta T. Virá do arquivo .dat
float tmax;         // Tempo máximo da simulação em segundos. Virá do arquivo .dat
float cdao;     // Virá do arquivo .dat

float r;
float a;        // Celeridade da onda
float qo;
float b;
float cm;
float cp;
float dx;       // Delta X. Comprimento do tudo pelo número de segmentos.
float tf;

float as;
float t;
float bp;
float bm;

float* HMatriz;         // Matriz dinâmica para H ( Pressao )
float* QMatriz;         // Matriz dinâmica para Q ( Vazao )
float* TMatriz;         // Matriz dinâmica para T ( Tempo )

float* d_HMatriz;         // Matriz dinâmica para H ( Pressao ) em device
float* d_QMatriz;         // Matriz dinâmica para Q ( Vazao ) em device
float* d_TMatriz;         // Matriz dinâmica para T ( Tempo ) em device

float* d_H;         // Matriz dinâmica para H ( Pressao ) que será alocada no device
float* d_Q;         // Matriz dinâmica para Q ( Vazao ) que será alocada no device
float* d_hp;         // Vetor dinâmico para hp, para auxílio no MOC que será alocada no device.
float* d_qp;         // Vetor dinâmico para qp, para auxílio no MOC que será alocada no device.

int i;          // Variável de controle nos laços for
int linha;      // Variável de controle nos laços for
int controle;   // Controle auxiliará para linkar o vetor auxiliar H e Q as respectivas matrizes.
int n;                  // Numeros de segmentos. Virá do arquivo .dat
int ns;         // Numero de segmentos + segmento de montante + segmento de jusante
```

```
int N;                  // Numeros de segmentos * Delta T

struct ConstantesPrograma{
         int ns;
         float qo;
         float hr;
         float r;
         float b;
         float t;
         float dt;
         float tmax;
         int controle;

         __device__ int get_ns() {return ns;}
         __device__ float get_qo() {return qo;}
         __device__ float get_hr() {return hr;}
         __device__ float get_r() {return r;}
         __device__ float get_b() {return b;}
         __device__ float get_t() {return t;}
         __device__ float get_dt() {return dt;}
         __device__ float get_tmax() {return tmax;}
         __device__ int get_controle() {return controle;}
};

__constant__ __device__ ConstantesPrograma ConstantDeviceContantes;

ConstantesPrograma *host_constantes;

unsigned int nblocks;          // Numero de blocos que vao ser chamados
unsigned int nthreads;          // Numero de threads por blocos que vao ser chamadas

const int threadsPerBlock = 512;


FILE *nomearqe;         // Nome do Arquivo de Entrada
FILE *nomearqs; // Nome do Arquivo de Saída
int gravarDisco = 0; // variavel de controle de saida de gravacao em disco das matrizes.

char str[80];         // Precisei desta variável para pegar dados do .dat do arquivo de entrada


/*  Funções  */

// Calcula o MOC em GPU
void calc_HQ_GPU();

// Inicializar elementos do vetor com zeros.
void initZero(float*, int);

// Imprimir Matriz
```

```
void printMatrix(float*, int, int);

struct timeval p0start,p1start, p1stop, totalt;
/*************************************************************
*                                                            *
*                   Função main no host                          *
*                                                            *
*************************************************************/
int main(int argc, char** argv)
{

         if (argc == 2) {
                  gravarDisco = atoi(argv[1]);
                  printf("\nAtivado modo impressão em arquivo.\n\n");
         }

         float tgpu, t_total;


         // Tempo 0 do programa. Para mensurar o tempo total da aplicação.
         gettimeofday(&p0start,NULL);

             int k = cutGetMaxGflopsDeviceId();
             cudaSetDevice(k);

         /*== Leitura dos dados */
         nomearqe = fopen ("caso5.dat" , "r");
            if (nomearqe == NULL) perror ("Erro ao abrir arquivo");
            else
            {
                  fscanf (nomearqe, "%s", str);
                  fscanf (nomearqe, "%f", &hr);
                  fscanf (nomearqe, "%s", str);
                  fscanf (nomearqe, "%f", &f);
                  fscanf (nomearqe, "%s", str);
                  fscanf (nomearqe, "%f", &l);
                  fscanf (nomearqe, "%s", str);
                  fscanf (nomearqe, "%f", &d);
                  fscanf (nomearqe, "%s", str);
                  fscanf (nomearqe, "%d", &n);
                  fscanf (nomearqe, "%s", str);
                  fscanf (nomearqe, "%f", &tmax);
                  fscanf (nomearqe, "%s", str);
                  fscanf (nomearqe, "%f", &a);
                  fscanf (nomearqe, "%s", str);
                  fscanf (nomearqe, "%f", &cdao);
                  fclose (nomearqe);
            }

         // Limpa contadores de tempo da CPU
```

```
        timerclear(&p1stop);
        timerclear(&p1start);

        // Calculo na GPU
        calc_HQ_GPU();



        // Obtem tempo de gasto na CPU em ms
        timersub(&p1stop,&p1start,&totalt);
        tgpu = 0.001*(totalt.tv_sec*1000000 + totalt.tv_usec);

        // Obtem tempo TOTAL do programaem ms
        timersub(&p1stop,&p0start,&totalt);
        t_total = 0.001*(totalt.tv_sec*1000000 + totalt.tv_usec);

        printf("Tempo de execucao da GPU =                  %f ms\n",tgpu);

        printf("GPU_TOTAL_%d\t\t%d\t\t %8.14f \n",threadsPerBlock,n,t_total);
        printf("GPU_%d\t\t%d\t\t %8.14f \n",threadsPerBlock,n,tgpu);
}

/*************************************************************
*                                                            *
*               Função que calcula o                         *
*               Regime Permanente Inicial e o                *
*               Regime Transiente no device                  *
*                                                            *
*************************************************************/

__global__ void calc_HQ_RegimeTransiente(float* HMatriz, float* QMatriz, float* TMatriz)
{
        // Identificador da thread
            int i = blockDim.x * blockIdx.x + threadIdx.x;

        // Uso de memória da thread.
        int indice = (int)(i % threadsPerBlock);

        float cp,bp,cm,bm = 0.0;
        // Número de sessões de cada delta T.
        // Passado por parâmetro.
        int ns = ConstantDeviceContantes.get_ns();
        // Vazão no tempo = 0.
        float qo = ConstantDeviceContantes.get_qo();

        float hr = ConstantDeviceContantes.get_hr();

        float r = ConstantDeviceContantes.get_r();

        float b = ConstantDeviceContantes.get_b();
```

```
float t = ConstantDeviceContantes.get_t();

float dt = ConstantDeviceContantes.get_dt();
// Tempo máximo da simulação.
float tmax = ConstantDeviceContantes.get_tmax();

// Controle
int controle = 0;

__shared__ float H_tempo_atual[threadsPerBlock];
__shared__ float Q_tempo_atual[threadsPerBlock];
__shared__ float H_proximo_tempo[threadsPerBlock];
__shared__ float Q_proximo_tempo[threadsPerBlock];

/*
Regime Permanente Inicial
*/

    if (i < ns) {
         H_tempo_atual[indice] = hr-(i)*r*qo*qo;
         Q_tempo_atual[indice] = qo;

         __syncthreads();

         HMatriz[(controle * ns) + i] = H_tempo_atual[indice];
         QMatriz[(controle * ns) + i] = Q_tempo_atual[indice];
         TMatriz[(controle * ns) + i] = t;
}

/*
Regime Transitório
*/
    if (i < ns) {
         while(t <= tmax) {
                  t = t + dt;
                  controle++;

              /*  Pontos Interiores  */
                  if (i < ns && i >= 1) {
                           cp = H_tempo_atual[indice-1]+b*Q_tempo_atual[indice-1];
                           bp = b+r*fabs(Q_tempo_atual[indice-1]);
                           cm = H_tempo_atual[indice+1]-b*Q_tempo_atual[indice+1];
                           bm = b+r*fabs(Q_tempo_atual[indice+1]);
                           Q_proximo_tempo[indice] = (cp-cm)/(bp+bm);
                           H_proximo_tempo[indice] = cp - bp * Q_proximo_tempo[indice];
                  }

              /*  Condicao de Contorno de montante: Reservatorio  */
                  if (i == 0) {
                           H_proximo_tempo[0] = hr;
                           cm = H_tempo_atual[1] - b * Q_tempo_atual[1];
```

```
                                bm = b + r * fabs(Q_tempo_atual[1]);
                                Q_proximo_tempo[0] = (hr-cm) /bm;
                        }
                    /*  Condicao de contorno de jusante: Válvula  */
                        if (i == ns-1) {
                                cp = H_tempo_atual[ns-2] + b * Q_tempo_atual[ns-2];
                                Q_proximo_tempo[ns-1] = 0;
                                H_proximo_tempo[ns-1] = cp;
                        }
                    /*  Atualizacao das variaveis e impressao  */

                        H_tempo_atual[indice] = H_proximo_tempo[indice];
                        Q_tempo_atual[indice] = Q_proximo_tempo[indice];

                        __syncthreads();
                    /*  Atualizacao na matriz  */
                        HMatriz[(controle * ns) + i] = H_tempo_atual[indice];
                        QMatriz[(controle * ns) + i] = Q_tempo_atual[indice];
                        TMatriz[(controle * ns) + i] = t;

                    } // Fim do laço for
        }
}


/************************************************************
*                                                           *
*    Função que calcula o MOC                               *
*    rodando trechos paralelizáveis em GPU                  *
*                                                           *
************************************************************/
void calc_HQ_GPU() {


gettimeofday(&p1start,NULL);

/* Calculo das Constantes */
        as = (PI*d*d)/4;
        dx = l/n;
        dt = dx/a;
        ns = n + 2; // Numero de segmentos + segmento de montante + segmento de jusante
        r = f * dx/(2*g*d*as*as);
        b = a/(g*as);

/* Alocação dos Vetores e Matrizes H e Q em host */

        linha = ((int)(tmax/dt))+1;
        N = ns + (ns * linha) + ns; // Delta T para 0 + conjunto de delta T + Última interação


        // Controle auxiliará para linkar o vetor auxiliar H e Q as respectivas matrizes.
```

```
        controle = 0;

            size_t sizeMATRIX = N * sizeof(float);

        HMatriz = (float*)malloc(sizeMATRIX);
        QMatriz = (float*)malloc(sizeMATRIX);
        TMatriz = (float*)malloc(sizeMATRIX);

        // Aloca vetores principais na memória do device.
            cutilSafeCall( cudaMalloc((void**)&d_HMatriz, sizeMATRIX) );
            cutilSafeCall( cudaMalloc((void**)&d_QMatriz, sizeMATRIX) );
            cutilSafeCall( cudaMalloc((void**)&d_TMatriz, sizeMATRIX) );

/* Inicialização de H e Q */

        initZero(HMatriz, N);
        initZero(QMatriz, N);
        initZero(TMatriz, N);

            cutilSafeCall( cudaMemcpy(d_TMatriz, TMatriz, sizeMATRIX, cudaMemcpyHostToDevice) );
            cutilSafeCall( cudaMemcpy(d_QMatriz, QMatriz, sizeMATRIX, cudaMemcpyHostToDevice) );
            cutilSafeCall( cudaMemcpy(d_HMatriz, HMatriz, sizeMATRIX, cudaMemcpyHostToDevice) );


/*  Calculo do Regime Permanente Inicial  */
/*  Inicio do transitorio  */
       t = 0;
            qo = sqrt(((cdao*cdao)*2*hr*hr)/(1+(cdao*cdao)*2*g*n*r));

        host_constantes = new ConstantesPrograma;

        host_constantes[0].ns               = ns;
        host_constantes[0].hr               = hr;
        host_constantes[0].r                = r;
        host_constantes[0].b                = b;
        host_constantes[0].dt               = dt;
        host_constantes[0].tmax             = tmax;
        host_constantes[0].controle         = controle;
        host_constantes[0].t                = t;
        host_constantes[0].qo               = qo;

        // Alocação das constantes no DEVICE com uso de CONSTANT memory
        cudaMalloc((void**)&ConstantDeviceCntantes, sizeof(ConstantesPrograma));
        cudaMemcpyToSymbol(ConstantDeviceContantes, host_constantes, sizeof(ConstantesPrograma));

        nthreads = threadsPerBlock;
        nblocks = (ns + threadsPerBlock ) / threadsPerBlock;

        gettimeofday(&p1start,NULL);
        calc_HQ_RegimeTransiente<<<nblocks, nthreads>>>(d_HMatriz, d_QMatriz, d_TMatriz);
```

```
        // Traz de volta vetores principais da memoria do device para a memoria do host
        cudaMemcpy(TMatriz , d_TMatriz, sizeMATRIX, cudaMemcpyDeviceToHost);
        cudaMemcpy(QMatriz , d_QMatriz, sizeMATRIX, cudaMemcpyDeviceToHost);
        cudaMemcpy(HMatriz , d_HMatriz, sizeMATRIX, cudaMemcpyDeviceToHost);


       gettimeofday(&p1stop, NULL);
    /*  Impressao da Matriz Completa  */
        if(gravarDisco == 1){
                nomearqs = fopen ("saida_vetor_MOC_GPU.dat" , "w");
                for(i=0 ; i<N ; i++) {
                            fprintf(nomearqs, "%4.2f   ", TMatriz[i]);
                            fprintf(nomearqs, "%4.5f   ", HMatriz[i]);
                            fprintf(nomearqs, "%4.5f \n", QMatriz[i]);
                }
                fclose (nomearqs);
        }

        free(HMatriz);
        free(QMatriz);
        free(TMatriz);

        cudaFree(d_HMatriz);
        cudaFree(d_QMatriz);
        cudaFree(d_TMatriz);

}

/*************************************************************
*                                                            *
*          Função que inicializa matrizes com 0.0            *
*                                                            *
*************************************************************/
void initZero(float* data, int n)
{
        for (int i = 0; i < n; ++i){
                data[i] = 0.0;
        }
}
```

A.3 versão OpenMP

```
/*************************************************************
*
*
*         Programa Metodo das caracteristicas com OpenMP
```

```
*
*
*          Orlando Saraiva Jr (saraiva(at)rc.unesp.br )
*          Prof. Vitor Rafael Coluci vitor(at)ft.unicamp.br
*
*        setembro/2011
*
*************************************************************/

#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>


#define PI 3.14159265
#define g 9.806

float hr;
float f;
float l;          // Tamanho horizontal do pipeline em metros.
float d;         // Diametro do tubo.
float dt;       // Delta T.
float tmax;         // Tempo máximo da simulação em segundos.
float cdao;

float r;
float a;        // Celeridade da onda
float qo;
float b;
float cm;
float cp;
float dx;       // Comprimento do tudo pelo número de segmentos.
float tf;

float as;
float t;
float bp;
float bm;

float* HMatriz;         // Matriz dinâmica para H ( Pressao )
float* QMatriz;         // Matriz dinâmica para Q ( Vazao )
float* TMatriz;         // Matriz dinâmica para T ( Tempo )

float* d_HMatriz;         // Matriz dinâmica para H ( Pressao ) em device
float* d_QMatriz;         // Matriz dinâmica para Q ( Vazao ) em device
float* d_TMatriz;         // Matriz dinâmica para T ( Tempo ) em device

float* H;         // Vetor dinâmico para H ( Pressao )
float* Q;         // Vetor dinâmico para Q ( Vazao )
```

```
float* hp;          // Vetor dinâmico para hp
float* qp;          // Vetor dinâmico para qp


float* d_H;          // Matriz dinâmica para H ( Pressao ) no device
float* d_Q;          // Matriz dinâmica para Q ( Vazao ) no device
float* d_hp;          // Vetor dinâmico para hp
float* d_qp;          // Vetor dinâmico para qp

int i;           // Variável de controle nos laços for
int linha;       // Variável de controle nos laços for
int controle;    // Controle auxiliará para linkar o vetor
                 // auxiliar H e Q as respectivas matrizes.
int n;                   // Numeros de segmentos
int ns;          // Numero de segmentos + montante + jusante

int N;                   // Numeros de segmentos * Delta T

FILE *nomearqe;          // Nome do Arquivo de Entrada
FILE *nomearqs; // Nome do Arquivo de Saída

int gravarDisco = 0;
int chunk = 5;

char str[80];

/*  Funções  */
// Calcula o MOC em CPU
void calc_HQ_CPU();

// Inicializar elementos do vetor com zeros.
void testAlocation(float*);

// Inicializar elementos do vetor com zeros.
void initZero(float*, int);

/*************************************************************
*                                                            *
*                   Função main no host                          *
*                                                            *
*************************************************************/
int main(int argc, char** argv)
{
         float tcpu, t_total;
             struct timeval p0start, p1start, p1stop, totalt;
         unsigned int timer;          //Variavel para contagem de tempo.

         // Tempo 0 do programa. Para mensurar o tempo total da aplicação.
         gettimeofday(&p0start,NULL);

         if(argc==3) {
```

```
                gravarDisco = atoi(argv[2]);
                chunk = atoi(argv[1]);
//                printf("\n# chunk = %d \n",chunk);
//                printf("\nAtivado modo impressão em arquivo.\n\n");
        }

        if(argc==2) {
                chunk = atoi(argv[1]);
//                printf("\n# chunk = %d \n",chunk);
        }
        /* = Leitura dos dados = */
        nomearqe = fopen ("caso5.dat" , "r");
           if (nomearqe == NULL) perror ("Erro ao abrir arquivo");
           else
           {
                 fscanf (nomearqe, "%s", str);
                      fscanf (nomearqe, "%f", &hr);
               fscanf (nomearqe, "%s", str);
                      fscanf (nomearqe, "%f", &f);
               fscanf (nomearqe, "%s", str);
                      fscanf (nomearqe, "%f", &l);
               fscanf (nomearqe, "%s", str);
                      fscanf (nomearqe, "%f", &d);
               fscanf (nomearqe, "%s", str);
                      fscanf (nomearqe, "%d", &n);
               fscanf (nomearqe, "%s", str);
                      fscanf (nomearqe, "%f", &tmax);
               fscanf (nomearqe, "%s", str);
                      fscanf (nomearqe, "%f", &a);
               fscanf (nomearqe, "%s", str);
                      fscanf (nomearqe, "%f", &cdao);
                      fclose (nomearqe);
        }

        // Limpa contadores de tempo da CPU

        timerclear(&p1stop);
        timerclear(&p1start);

        // Calculo na CPU (serial)
        gettimeofday(&p1start,NULL);

        calc_HQ_CPU();

        gettimeofday(&p1stop, NULL);

        // Obtem tempo de gasto na CPU em ms
        timersub(&p1stop,&p1start,&totalt);
        tcpu = 0.001*(totalt.tv_sec*1000000 + totalt.tv_usec);

        // Obtem tempo TOTAL do programa em ms
```

```
        timersub(&p1stop,&p0start,&totalt);
        t_total = 0.001*(totalt.tv_sec*1000000 + totalt.tv_usec);

        printf("Tempo de execucao da CPU =                  %f ms\n",tcpu);

        printf("OMP_TOTAL_%d\t\t%d\t\t %8.14f \n",chunk,n,t_total);
        printf("OMP_%d\t\t%d\t\t %8.14f \n",chunk,n,tcpu);

}

/*************************************************************
*                                                            *
*    Função que calcula o MOC no host e de forma SERIAL      *
*                                                            *
*************************************************************/
void calc_HQ_CPU() {

        if(gravarDisco == 1){
                nomearqs = fopen ("saida_vetor_MOC_OMP.dat" , "w");
        }

/*= Calculo das Constantes ==*/
        as = (PI*d*d)/4;
        dx = l/n;
        dt = dx/a;
        ns = n + 2;
        r = f * dx/(2*g*d*as*as);
        b = a/(g*as);

/*= Alocação dos Vetores e Matrizes H e Q */

        linha = ((int)(tmax/dt))+1;
        N = ns + (ns * linha) + ns;
/*        printf("Numero de elementos da matriz =          %d \n",N);
        printf ("linha = %d\n",linha);
        printf ("tmax = %4.2f\n",tmax);
        printf ("dt = %4.2f\n",dt);
        printf ("n = %d\n",n);
        printf ("ns = %d\n",ns);
        printf ("N = %d\n",N);
*/
        // Controle para linkar o vetor auxiliar H e Q as respectivas matrizes.
        controle = 0;

            size_t sizeMATRIX = N * sizeof(float);
            size_t sizeVECTOR = ns * sizeof(float);

        HMatriz = (float*)malloc(sizeMATRIX);
        QMatriz = (float*)malloc(sizeMATRIX);
        TMatriz = (float*)malloc(sizeMATRIX);
```

```
        H = (float*)malloc(sizeVECTOR);
        Q = (float*)malloc(sizeVECTOR);
        hp = (float*)malloc(sizeVECTOR);
        qp = (float*)malloc(sizeVECTOR);

        testAlocation(HMatriz);
        testAlocation(QMatriz);
        testAlocation(TMatriz);

        testAlocation(H);
        testAlocation(Q);
        testAlocation(hp);
        testAlocation(qp);

/*== Inicialização de H e Q ==*/

        initZero(HMatriz, N);
        initZero(QMatriz, N);
        initZero(TMatriz, N);

        initZero(H, ns);
        initZero(Q, ns);
        initZero(hp, ns);
        initZero(qp, ns);
       t = 0;
/*= Calculo do Regime Permanente Inicial */

            qo=sqrt(((cdao*cdao)*2*hr*hr)/(1+(cdao*cdao)*2*g*n*r));

        #pragma omp parallel shared(H,Q,HMatriz,QMatriz,TMatriz,hr,r,qo,t,ns,chunk) private(i)
        {
                #pragma omp for schedule(dynamic,chunk) nowait
                    for(i=0; i<= ns ; i++) {
                         H[i] = hr-(i)*r*qo*qo;
                         Q[i] = qo;
//                         printf ("H[%d] = %4.2f   \n",i,H[i]);
                         HMatriz[i] = H[i];
                         QMatriz[i] = Q[i];
                         TMatriz[i] = t;
                    }
        }

/*  Inicio do transitorio  */
        while(t <= tmax) {
                t=t+dt;
                controle++;

             /*  Pontos Interiores  */
                #pragma omp parallel shared(H,Q,hp,qp,r,b,ns,chunk) private(i,cm,bm,cp,bp)
                {
                        #pragma omp for schedule(dynamic,chunk) nowait
```

```
                    for(i=1 ; i<ns ; i++){
                            cp = H[i-1]+b*Q[i-1];
                            bp = b+r*fabs(Q[i-1]);
                            cm = H[i+1]-b*Q[i+1];
                            bm = b+r*fabs(Q[i+1]);
                            qp[i] = (cp-cm)/(bp+bm);
                            hp[i] = cp-bp*qp[i];
                    }
            }

         /*  Condicao de Contorno de montante: Reservatorio  */
            hp[0] = hr;
            cm = H[1]-b*Q[1];
            bm = b+r*fabs(Q[1]);
            qp[0] = (hr-cm)/bm;

         /*  Condicao de contorno de jusante: Válvula  */
            cp = H[ns-2]+b*Q[ns-2];
            qp[ns-1] = 0;
            hp[ns-1] = cp;

        /*  Atualizacao das variaveis e impressao  */
            #pragma omp parallel shared(H,Q,hp,qp,ns,chunk) private(i)
            {
                    #pragma omp for schedule(dynamic,chunk) nowait
                    for(i=0 ; i<ns ; i++) {
                            H[i] = hp[i];
                            Q[i] = qp[i];
                    }
            }
        /*   Atualizacao na matriz   */
            #pragma omp parallel shared(H,Q,HMatriz,QMatriz,TMatriz,ns,t,chunk) private(i
            {
                    #pragma omp for schedule(dynamic,chunk) nowait
                    for(i=0 ; i<=ns ; i++) {
                            HMatriz[(controle * ns ) + i] = H[i];
                            QMatriz[(controle * ns ) + i] = Q[i];
                            TMatriz[(controle * ns ) + i] = t;
                    }
            }
         } // Fim do laço for

/*   Impressao da Matriz Completa   */
    if(gravarDisco == 1){
            for(i=0 ; i<N ; i++) {
                    fprintf(nomearqs, "%4.2f \t",  TMatriz[i] );
                    fprintf(nomearqs, "%4.5f \t", HMatriz[i] );
                    fprintf(nomearqs, "%4.5f\n", QMatriz[i]);
            }
    }
```

```
        free(HMatriz);
        free(QMatriz);
        free(TMatriz);

        free(H);
        free(Q);
        free(hp);
        free(qp);

        if(gravarDisco == 1){
                fclose (nomearqs);
        }
}

/*************************************************************
*                                                            *
*          Função que inicializa matrizes com 0.0              *
*                                                            *
*************************************************************/
void testAlocation(float* data)
{
        if(data == 0 ) {
                printf("Falha na alocação\n");
                printf("Memoria RAM indisponivel para os parametros passados\n");
                exit(-1);
        }
}

/*************************************************************
*                                                            *
*          Função que inicializa matrizes com 0.0              *
*                                                            *
*************************************************************/
void initZero(float* data, int n)
{
        int i;
        for (i = 0; i < n; ++i){
                data[i] = 0.0;
        }
}
```

A.4 Para comparar resultados obtidos.

```
/*************************************************************
*                Comparar resultados                         *
*                entre versao SERIAL e                       *
*                PARALELA                                    *
```

```
*                                                                     *
*                                                                     *
*          Orlando Saraiva Jr (saraiva(at)rc.unesp.br )             *
*          Prof. Vitor Rafael Coluci vitor(at)ft.unicamp.br         *
*                                                                     *
*        setembro/2011                                                *
*                                                                     *
**********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <string.h>

FILE *nomearq_01;          // Nome do primeiro arquivo
FILE *nomearq_02;          // Nome do segundo arquivo

double margem_de_erro = 1e-6;         // Diferença máxima
                                    // aceita entre os resultados.

float T_01, H_01 , Q_01;
float T_02, H_02 , Q_02;


void comparar(char [],char []);

/*************************************************************
*                                                             *
*                   Função main no host                        *
*                                                             *
*************************************************************/
int main(int argc, char** argv)
{

         // Comparar Resultados dos vetores

         char * arq1;
         char * arq2;

         arq1 = (char *)"saida_vetor_MOC_serial.dat";
/*
         printf("\n\n Confrontando SERIAL com versao PASCAL.\n\n");
         arq2 = (char *)"saidaP.DAT";
         comparar(arq1,arq2);

         printf("\n\nConfrontando SERIAL com versao OpenMP.\n\n");
         arq2 = (char *)"saida_vetor_MOC_OMP.dat";
         comparar(arq1,arq2);

*/
```

```
        printf("\n\nConfrontando SERIAL com versao GPU.\n\n");
        arq2 = (char *)"saida_vetor_MOC_GPU.dat";
        comparar(arq1,arq2);

}
/*************************************************************
*                                                            *
*                  Comparar dois arquivos                        *
*                                                            *
*************************************************************/
void comparar(char arq1[],char arq2[]) {

        // Abertura dos arquivos .dat
        nomearq_01 = fopen (arq1 , "r");
        if (nomearq_01 == NULL) {
                printf("Erro ao abrir primeiro arquivo.  ");
                return;
        }
        nomearq_02 = fopen (arq2 , "r");
        if (nomearq_02 == NULL) {
                printf("Erro ao abrir segundo arquivo.  ");
                return;
        }

        //  Arquivo 01 x Arquivo 02
        int controle_linha = 0;
        double erroH, erroQ = 0;

        while (fscanf(nomearq_01, "%f %f %f", &T_01 , &H_01 , &Q_01) != EOF) {
                fscanf(nomearq_02, "%f %f %f", &T_02 , &H_02 , &Q_02);
                controle_linha++;

                if (fabs(H_01 - H_02) > margem_de_erro){
                        erroH = erroH + pow (fabs(H_01 - H_02),2);
                        printf("n = %d ",controle_linha);
                        printf("erro = %4.4f ",fabs(H_01 - H_02));
                        printf("erro^2 =  %4.4f \n",pow (fabs(H_01 - H_02),2) );
                        printf("SumErro =  %4.4f \n" ,erroH);
                }

                if (fabs(Q_01 - Q_02) > margem_de_erro) {
                        erroQ = erroQ +  pow (fabs(H_01 - H_02),2);
                }
        }

        printf("H = %4.4f \n",erroH);
        printf("Q = %4.4f \n",erroQ);
        printf("n = %d \n\n",controle_linha);

        printf("EQM (H) = %4.4f \n",erroH/controle_linha);
```

```
        printf("EQM (Q) = %4.4f \n",erroQ/controle_linha);
        printf("Margem de erro: %4.8f \n",margem_de_erro);


        fclose(nomearq_01);
        fclose(nomearq_02);
        printf("%s \n","OK");
}
```

Trabalhos Publicados Pelo Autor

1. Orlando Saraiva do Nascimento Júnior, Lubienska Cristina L. Jaquiê Ribeiro, Vitor Rafael Coluci . "Computação Paralela aplicada ao Método das Características em Sistemas Hidráulicos". *II Escola Regional de Alto Desempenho de São Paulo* (ERAD-SP 2011), São José dos Campos, São Paulo, Brasil, pg. 163-175, Julho 2011.

2. Orlando Saraiva do Nascimento Júnior. "Manual de Instalação da arquitetura CUDA no Sistema Operacional Ubuntu 10.10". *Documento interno* Departamento de Estatística, Matemática Aplicada e Computação. Unesp - Rio Claro, Brasil, Julho 2011.

3. Orlando Saraiva do Nascimento Júnior, Vitor Rafael Coluci . "Ganhos Computacionais com uso da Arquitetura CUDA na Resolução do Método das Características em Sistemas Hidráulicos". *III Escola Regional de Alto Desempenho de São Paulo* (ERAD-SP 2012), Campinas, São Paulo, Brasil, pg. 72-75, Julho 2012.