

**Universidade de São Paulo**  
**Instituto de Matemática e Estatística**  
**Bacharelado em Matemática Aplicada e Computacional com Habilitação em**  
**Mecatrônica e Sistemas Mecânicos**

**Tiago Fernandes D'Agostino**

**Implementação de solução computacional de múltiplas  
arquiteturas para o transitório hidráulico com interação fluído-  
estrutura**

**São Paulo**

**2020**



**Implementação de solução computacional de múltiplas arquiteturas para o transitório hidráulico com interação fluído-estrutura**

Monografia final da disciplina MAP2060 – Trabalho de Formatura.

Dissertação apresentada ao Programa de Graduação em Matemática Aplicada Computacional com habilitação em Engenharia Mecatrônica da Unidade IME, Universidade de São Paulo, como parte dos requisitos para a obtenção do título de Bacharel em Matemática Aplicada e Computacional.

Orientador: Prof. Dr. Thiago de Castro Martins

**São Paulo**

**2020**

## **Resumo**

A interação fluido-estrutura (IFE) é um importante fenômeno físico que ocorre em sistemas de transporte de fluidos. A superfície sólida da estrutura interage com o fluido, gerando transientes hidráulicos que podem resultar em problemas como ruído contínuo, vibrações e até mesmo a ruptura da estrutura. Como esses transitórios são regidos por equações diferenciais, uma maneira de prever essas situações de risco é através de simulações computacionais, com o uso de resoluções numéricas.

Com o intuito de reduzir o tempo de processamento, deseja-se comparar o desempenho entre simulações do transiente hidráulico em diferentes arquiteturas computacionais, usando técnicas de programação convencional (ANSI C), paralela (OpenMP) e em placa de vídeo (CUDA).

Os resultados obtidos destacam um melhor desempenho no processamento em placa de vídeo, e também reforçam que é preciso ter conhecimento da arquitetura da CPU e/ou GPU para que o algoritmo seja adaptado de forma a extrair a melhor performance possível do hardware.

## **Abstract**

Fluid-structure interaction (FSI) is an important physical phenomenon that occurs in fluid transport systems. The structure interacts with the fluid, causing hydraulic transients that can result in problems such as noise, vibrations and even the rupture of the structure. As these transients are ruled by differential equations, one way to predict these risky situations is through computer simulations, using numerical methods.

Using conventional (ANSI C), parallel (OpenMP) and graphic card (CUDA) programming techniques, it is wanted to compare hydraulic transient simulations performance in different computational architectures.

The achieved results highlight a better performance of the graphic card processing, and also shows how necessary is to know the CPU / GPU architecture, so that the algorithm can be adapted in order to obtain the best performance from the hardware.

## **Sumário**

1	INTRODUÇÃO .....	1
1.1	Motivação .....	1
1.2	Objetivo .....	4
1.3	Fundamentação Teórica .....	4
1.3.1	Transiente Hidráulico.....	4
1.3.2	Modelagem do Problema .....	8
1.3.3	Metodologia de Cálculos .....	10
1.3.4	Arquitetura Computacional e Computação Paralela.....	14
1.3.5	Arquitetura GPU / CUDA.....	17
1.3.6	<i>OpenMP</i> e por que usá-lo .....	22
1.3.7	Métricas para análise de desempenho.....	22
2	DESENVOLVIMENTO .....	23
2.1	Pseudocódigo.....	23
2.2	Implementação .....	25
2.3	Desenvolvimento Serial .....	26
2.4	Desenvolvimento Paralelo.....	30
2.5	Desenvolvimento CUDA .....	34
3	SIMULAÇÃO E RESULTADOS.....	44
3.1	Cenário com detalhamento de 1 metro .....	44
3.2	Cenário com detalhamento de 10 centímetros.....	47
4	CONCLUSÃO .....	49

5	APÊNDICE .....	51
6	REFERÊNCIAS BIBLIOGRÁFICAS.....	82

## **Lista de Figuras**

Figura 1: Eventos de Golpe de aríete reportados por ano.....	2
Figura 2: Quantitativo total de eventos reportados.....	2
Figura 3: Percentual do total de eventos reportados por tipo de evento .....	2
Figura 4: Formação de ondas após uma perturbação no sistema hidráulico [9].....	6
Figura 5: Ilustração do sistema simulado (Reservatório – Tubo – Válvula) .....	8
Figura 6: 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.....	9
Figura 7: 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).....	14
Figura 8: Arquitetura de máquinas de <i>Von Neumann</i> .....	15
Figura 9: Visão geral de um sistema de memória .....	16
Figura 10: Visão geral de um sistema de memória compartilhada.....	17
Figura 11: Visão geral da arquitetura GPU.....	18
Figura 12: Exemplo de <i>thread</i> , bloco, <i>grid</i> e <i>kernel</i> . [16] .....	20
Figura 13: Hierarquia de memórias em CUDA. [16] .....	20
Figura 14: Pseudocódigo que implementa o método das características.....	24
Figura 15: Trecho de código do passo 5 em desenvolvimento serial.....	27
Figura 16: Trecho de código do passo 6 no desenvolvimento serial.....	27
Figura 17: Trecho de código do passo 7 no desenvolvimento serial.....	29

Figura 18: Evolução computacional do passo 7, em processamento serial, onde o cálculo de pressão e vazão de um segmento (em azul) depende do valor de segmentos em tempo anterior (em vermelho).....	30
Figura 19: Trecho de código do passo 6 no desenvolvimento paralelo.....	31
Figura 20: Trecho de código do passo 7 no desenvolvimento paralelo.....	33
Figura 21: Evolução computacional do passo 7, em processamento <i>multicore</i> , onde ocorre o cálculo de dois segmentos simultaneamente. ....	34
Figura 22: Trecho de código que exibe a utilização de malloc, cudaMalloc e cudaMemcpy ...	35
Figura 23: Evolução do cálculo executado no passo 7, implementado em CUDA, onde o cálculo de pressão e vazão ocorre em todos segmentos ímpares (rosa) de um tempo simultaneamente, e em seguida são calculados todos os segmentos pares (verde) do mesmo tempo. ....	37
Figura 24: Trecho de código do passo 7 na implementação CUDA .....	38
Figura 25: Funcionamento do ponto de sincronização. Em (a), em destaque, os segmentos que serão trabalhados até a próxima sincronização. Em (b), a evolução do processamento de duas threads para o cálculo dos segmentos 1 e 2 (t0), 3 e 4 (t1) e 0 (t1). Em (c) ocorre o ponto de sincronização e o processamento só continua quando todos os segmentos da linha 1, daquele bloco, foram calculados.....	39
Figura 26: Trecho de código que encontra o número de blocos a ser utilizado .....	40
Figura 27: Sequência de cálculos realizados em blocos na implementação CUDA. Após a seleção do bloco, ocorre o cálculo dos segmentos, usando sincronização. Em seguida, um novo bloco é escolhido de forma que contenha a última coluna de segmentos do bloco anterior, para, assim, os segmentos serem calculados. ....	41
Figura 28: Trecho de código que exibe aplicação da <i>shared memory</i> .....	42
Figura 29: Trecho de código que exibe aplicação da memória constante .....	43



## **Lista de Tabelas**

Tabela 1: Parâmetros dos problemas simulados.....	25
Tabela 2: Parâmetros que devem ser informados para execução do programa.....	25
Tabela 3: Parâmetros de entrada da simulação dos problemas com granularidade 1m .....	45
Tabela 4: Tempos medidos na simulação do cenário om granularidade de 1m .....	46
Tabela 5: Cálculos de Speedup para o cenário de granularidade 1m .....	46
Tabela 6: Parâmetros de entrada da simulação dos problemas com granularidade 10cm.....	47
Tabela 7: Tempos medidos na simulação do cenário om granularidade de 10cm .....	48
Tabela 8: Cálculos de Speedup para o cenário de granularidade 10cm .....	48

# 1 INTRODUÇÃO

## 1.1 Motivação

Este trabalho se aplica sobre o problema do transitório hidráulico que pode ocorrer em instalações de transporte de fluídos. Em grandes instalações esses transitórios são responsáveis por carregamentos hidrodinâmicos e mecânicos consideráveis (pressões, vibrações, forças e deslocamentos) que, principalmente em instalações de natureza crítica (como instalações petrolíferas e nucleares), devem ser previstos com o intuito de se fazer um dimensionamento seguro, mas também sem deixar o fator econômico de lado.

Deste modo, existe uma significativa importância de se abordar a operação de sistemas hidráulicos de transporte de fluídos em múltiplas aplicações críticas do ponto de vista econômico, ambiental e humano, além de também conhecer as condições extremas de operação que ocorrem durante os transientes hidráulicos (Martelo Hidráulico), isto é, quando há bruscas alterações do movimento do fluido.

Tamanha é a importância desse objeto de estudo que a Electric Power Research Institute, (EPRI) realizou um ensaio onde compilou eventos relacionados ao efeito do transitório hidráulico e os classificou quanto ao tipo de planta em que ocorreu e a perturbação causadora do efeito. Nesse estudo foram analisadas as informações de 283 eventos relacionados ao transitório hidráulico em plantas de reatores nucleares do tipo BWR (*Boiling Water Reactor* ou Reator de Água Fervente) e PWR (*Pressured Water Reactor* ou Reator de Água Pressurizada) nos Estados Unidos entre 1969 e 1988. Esse trabalho [1] resultou num banco de dados com informações sobre a ocorrência do fenômeno, como pode ser observado nas Figura 1, Figura 2 Figura 3.

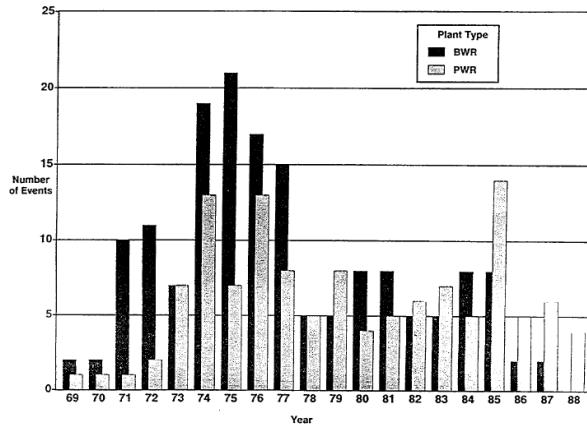


Figura 1: Eventos de Golpe de aríete reportados por año

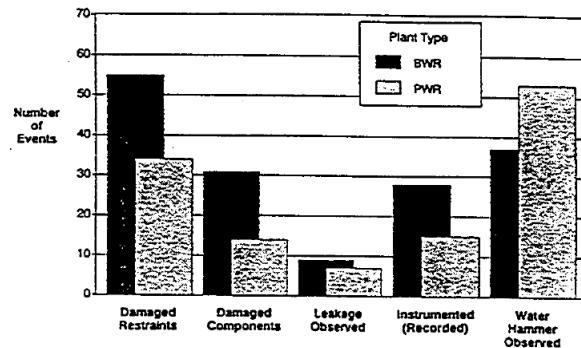


Figura 2: Quantitativo total de eventos reportados

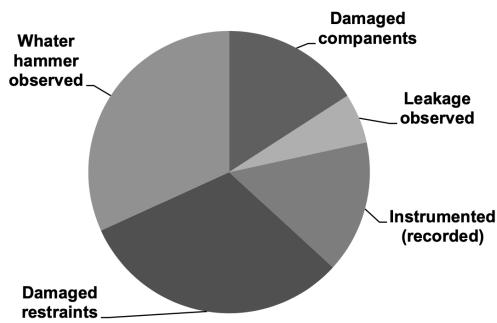


Figura 3: Percentual do total de eventos reportados por tipo de evento

Essa pesquisa fundamentou a impescindibilidade de se estudar devidamente o fenômeno a fim de prover ferramentas que investigue suas causas e possivelmente prevenir e mitigar as implicações do martelo hidráulico ou mesmo, se viável, eliminá-los.

Como o fenômeno do transiente hidráulico é regido por equações diferenciais, uma possibilidade de se investigar seu efeito é por meio de simulação computacional com o uso de resoluções numéricas. Entretanto, dependendo da complexidade e do tamanho do sistema simulado (comprimento de condutores, número de válvulas, tempo desejado de simulação etc), a resolução das equações diferenciais pode se tornar onerosa computacionalmente.

Para tentar reduzir o tempo de processamento das simulações, existem duas maneiras. Uma delas é o investimento em unidades de centrais de processamento (CPU) de maior desempenho, que não é influenciado pela técnica de programação utilizada. Uma outra maneira é se beneficiar da arquitetura computacional já existente no parque, implementando técnicas de programação paralela. Uma dessas técnicas de computação paralela é baseada em unidades de processamento gráfico [2].

O uso de GPU em simulações de sistemas hidráulicos tem exposto ganhos de desempenho convincentes, como na utilização de redes neurais artificiais paralelas para extração de conhecimento hidráulico de grandes sistemas de distribuição de água [3]. Em [4] também é enfatizado o potencial para ganhos na velocidade de resolver equações hidráulicas de distribuição de água.

Visto que o fenômeno do martelo hidráulico está associado aos efeitos mais destrutivos dos regimes transitórios em pressão e deve ser levado em conta no projeto de instalações hidráulicas [5], e com a crescente popularização das GPUs, além da queda do seu custo, considerando também as opções de locação desses equipamentos e *cloud computing* (computação em nuvem), surgiu a motivação para a realização deste trabalho.

## 1.2 Objetivo

Após a aquisição de maior conhecimento do fenômeno do transiente hidráulico, através do estudo do desenvolvimento histórico do problema e também dos métodos de solução existentes, espera-se com o desenvolvimento deste trabalho a construção de um modelo numérico pelo método das características para a solução do conjunto de equações do modelo.

Da mesma forma, pretende-se realizar uma comparação entre simulações do transiente hidráulico em diferentes arquiteturas computacionais, usando técnicas de programação paralela, visando obter uma configuração que diminua o tempo de processamento dos cálculos do modelo.

## 1.3 Fundamentação Teórica

### 1.3.1 Transiente Hidráulico

Instalação hidráulica é o conjunto de dispositivos e condutos destinados ao transporte de fluidos. Esses dispositivos são hidromecânicos (p.e. válvula, turbinas e bombas) e possuem a função de controlar o escoamento do fluido e realizar a transformação da energia mecânica em hidráulica e vice e versa [6].

Os dispositivos hidromecânicos executam algumas ações, ou manobras, para que o fluxo de escoamento do fluido na instalação seja controlado, como por exemplo a abertura e o fechamento de válvulas ou o acionamento de bombas. O movimento do fluido interage com a superfície sólida da estrutura da instalação hidráulica, gerando esforços mecânicos e deslocamentos para ambos. Essa interação dos movimentos do fluido e da instalação hidráulica, que devem ser analisadas simultaneamente e de forma acoplada, é denominada Interação Fluido Estrutura (IFE).

A IFE ocorre em qualquer sistema de tubulação para transporte de fluidos e para sistemas simples, geralmente, sua repercussão é diminuta e pode ser desconsiderada.

Entretanto, para algumas situações, em geral projetos de grande porte, os efeitos da IFE podem causar bruscas variações de pressão e vazão, provocando extensas consequências para o sistema hidráulico, como ruídos, vibrações e até ruptura da estrutura.

Sempre que ocorrem mudanças na quantidade de movimento do fluido ou na estrutura da tubulação através de variações de pressão ou vazão no sistema, surge um efeito chamado transitório, ou transiente, hidráulico. Ou seja, transitório hidráulico é o regime variado que ocorre durante a passagem de um regime permanente para outro regime permanente.

De acordo com [7], os transitórios hidráulicos podem ser definidos como “a situação de escoamento variável (no qual as variáveis associadas oscilam no tempo) existente entre duas condições de escoamento inicial e final, de regime permanente”.

Assim, qualquer alteração no movimento de um elemento do sistema, seja ocasionado por uma ação planejada, como acionamento de dispositivos hidromecânicos, ou acidental, como rupturas ou interferências externas, dão origem a esses fenômenos transitórios.

Num transitório hidráulico, as ondas de pressão geradas no sistema, sejam positivas ou negativas, podem ocorrer em amplitude e frequências em tal proporção que são capazes de acarretar no rompimento da tubulação ou falha nos dispositivos da instalação [8].

As variações de pressão ao longo da tubulação no transiente hidráulico ocorrem de maneira tão brusca que provocam sons similares a pancadas. Por isso, o transitório hidráulico também é comumente chamado de Golpe de Aríete.

A Figura 4 a seguir ilustra as ondas geradas com as variações de pressão num sistema hidráulico:

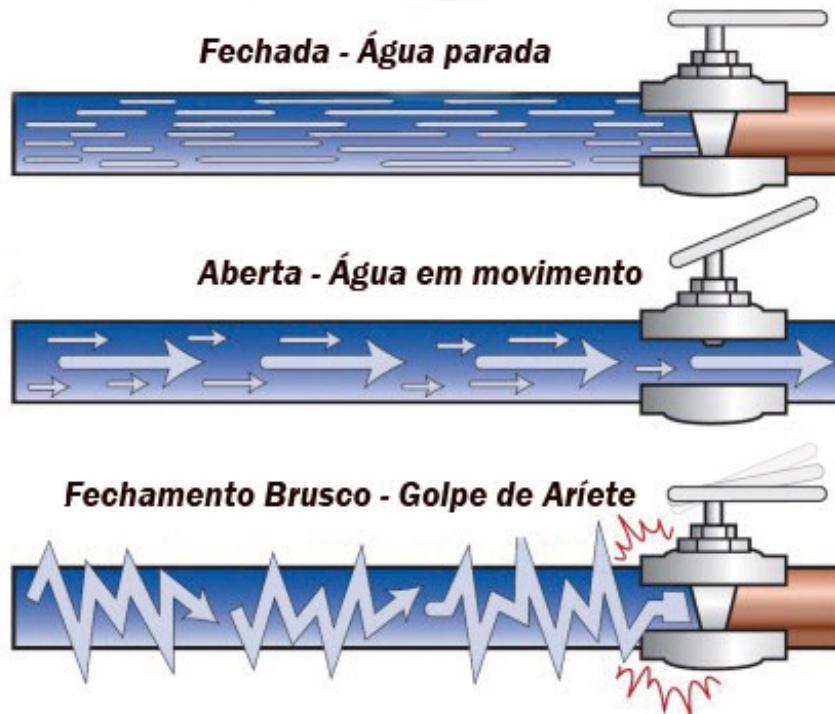


Figura 4: Formação de ondas após uma perturbação no sistema hidráulico [9]

Em sistemas onde a segurança é um fator crítico, como em usinas e grandes dutos de transporte de fluidos, a IFE deve ser muito bem analisada para que seja possível evitar qualquer consequência de efeitos transitórios.

As equações que descrevem o comportamento da vazão e da pressão do fluido em diferentes partes do sistema hidráulico e em diferentes instantes de tempo são as equações diferenciais parciais do movimento e da continuidade. E através de simulação computacional, onde equações diferenciais são resolvidas numericamente, é possível analisar os efeitos dos transitórios hidráulicos.

Existem alguns métodos para resolver essas equações de forma numérica, como o método das diferenças finitas, métodos dos elementos finitos, métodos das características, dentre outros [10] [11]. Ressaltando, no entanto, que a precisão do cálculo dependente vigorosamente da exatidão dos dados de entrada, como coeficientes das propriedades e condições iniciais do fluido e características gerais dos dispositivos hidromecânicos do sistema

(como material e forma). Obter esses valores, na prática, pode ser desafiador em sistemas de grande porte e alta complexidade.

Para a investigação numérica de fenômenos envolvendo transientes hidráulicos, no Brasil, a norma NBR 12215 [12] recomenda o método das características, pois trata-se de uma técnica numérica de solução de equações derivadas parciais sem solução analítica, como no caso dos transientes.

No método das características o intuito é transformar as equações diferenciais parciais em equações diferenciais ordinárias, através da discretização do sistema, para que sejam mais fáceis de resolver numericamente. As variáveis dependentes (como pressão) são calculadas e armazenadas em pontos de uma malha de variáveis independentes (como tempo e espaço), e as equações obtidas pelo método das características são transformadas em equações de diferenças finitas.

A resolução de tais equações pode ser computacionalmente onerosa dependendo do tamanho do sistema hidráulico que está sendo simulado ou do intervalo que deve ser feita uma nova simulação. Isso porque em cada iteração da simulação são calculados valores de pressão e vazão em cada ponto da malha.

Uma maneira de reduzir o custo computacional é adquirindo CPUs mais rápidas para processamento local, entretanto essa abordagem nem sempre é adotada, pois exige investimento em outros itens de infraestrutura que acompanha a adoção de equipamentos de alto desempenho, como salas especiais, adaptação do sistema energético, refrigeração adequada, segurança, podendo tornar essa solução proibitiva.

Uma outra maneira de reduzir o custo computacional é aproveitar ao máximo os recursos já existentes através da modificação do código computacional, explorando técnicas de computação paralela, como as baseadas em unidades de processamento gráfico [2]. As GPUs têm-se mostrado como um hardware versátil que se adapta facilmente ao parque de

infraestrutura já existente, adicionando robusta capacidade de processamento em cálculos numéricos.

Um modelo de computação heterogêneo utiliza a CPU para a parte sequencial da aplicação e a GPU fica encarregado de acelerar a parte da computacional paralela [2]. Ao passo que as CPUS modernas possuem em média 8 núcleos e no máximo 64, uma GPU é formada por milhares de núcleos, apoiados em uma arquitetura construída propriamente para processamento paralelo e em algoritmos escritos usando técnicas de computação paralela.

Uma fabricante vanguardista de GPUs é a NVIDIA, que criou um conjunto de ferramentas de *software* para facilitar a integração ao seu *hardware*, o CUDA. A linguagem de programação CUDA-C, uma versão estendida da linguagem C-ANSI [13] e do padrão aberto OpenCL, pode ser utilizada para programação paralela em sistemas heterogêneos [14].

O presente trabalho realiza uma abordagem computacional sobre este problema com o foco computacional e utiliza a arquitetura CUDA e a API de paralelismo OpenMP [15].

### 1.3.2 Modelagem do Problema

Para as simulações desse trabalho, escolheu-se um sistema hidráulico simples, mas muito comum nas construções, composto por um reservatório, um tubo e uma válvula. O nó montante do tubo estando conectado ao reservatório, e o nó jusante ligado na válvula, como pode ser observado na Figura 5.

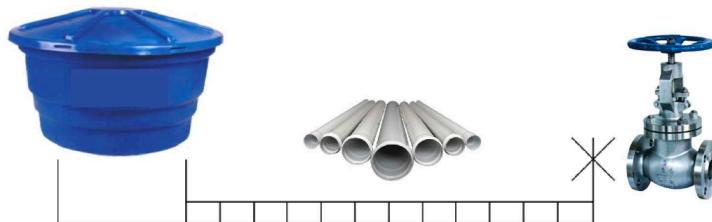


Figura 5: Ilustração do sistema simulado (Reservatório – Tubo – Válvula)

Nesse sistema, quando a válvula está aberta, um regime permanente inicial é estabelecido a partir do momento em que ocorre um fluxo contínuo de fluido no tubo. O transiente hidráulico surge quando a válvula é fechada, interrompendo o fluxo laminar, provocando ondas de variação de pressão no interior do tubo. Essas ondas oscilam até o que a movimentação interna do fluido entra em novo regime permanente, chamado regime permanente final.

O histórico da variação da pressão interna do tubo pode ser observado na Figura 6, onde dois segmentos distintos do tubo demonstram trechos sob ondas de sobrepressão, na cor vermelha, e trechos sob ondas de subpressão, na cor branca, em 6 instantes de tempo. Os trechos na cor azul representam um valor de pressão igual a pressão no regime permanente inicial. [16] Ainda na Figura 6, note, nos gráficos, que a variação de pressão oscila entre sobrepressão e subpressão, com picos que vão diminuindo e tendem a estabilidade em um novo regime permanente.

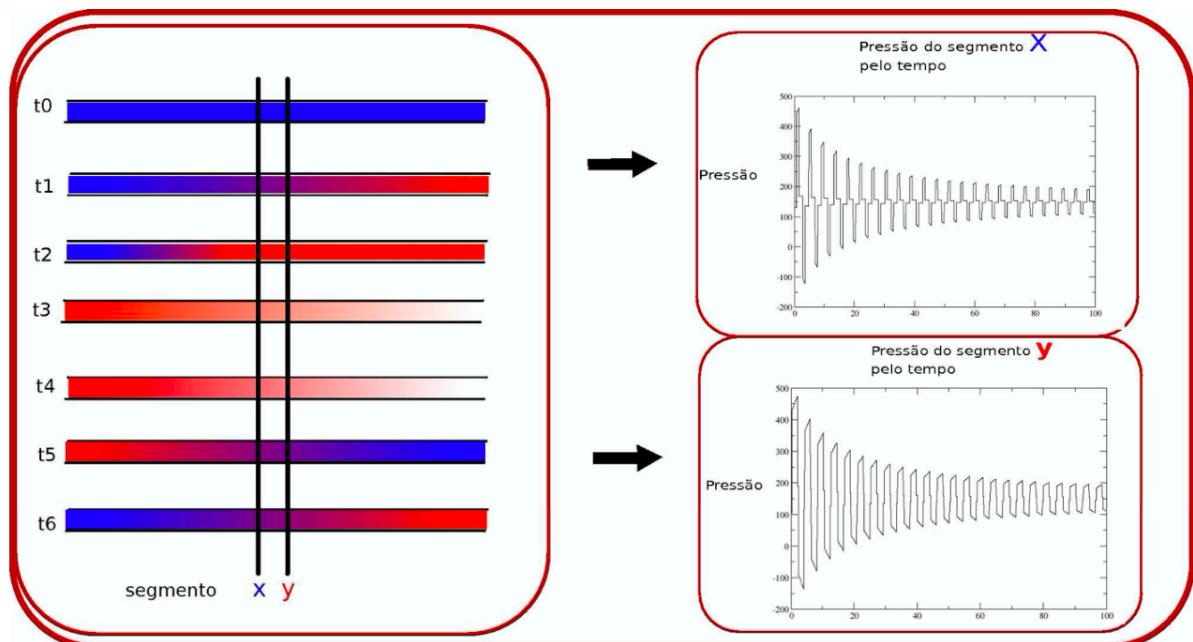


Figura 6: Ilustração do histórico da Pressão nos segmentos X e Y ao longo do tempo. Ao lado esquerdo, a visualização de ondas de sobrepressão e subpressão e os dois segmentos observados. Ao lado direito, dois gráficos ilustram o histórico da pressão nos dois segmentos - X e Y - em destaque. Por ser uma ilustração, desconsidera a vaporização que ocorreria nos valores apresentados.

### 1.3.3 Metodologia de Cálculos

As equações da quantidade de movimento (1) e da continuidade (2), que descrevem o comportamento do fluido, formam um par de equações diferenciais parciais hiperbólicas com duas variáveis dependentes, velocidade do fluido ( $V$ ) e carga piezométrica ( $H$ ), e duas variáveis independentes, distância ao longo da tubulação ( $x$ ) e tempo ( $t$ ) [6].

$$g \frac{\partial H}{\partial x} + V \frac{\partial V}{\partial x} + \frac{\partial V}{\partial t} + \frac{fV|V|}{2D} = 0 \quad (1)$$

$$V \frac{\partial H}{\partial x} + \frac{\partial H}{\partial t} + \frac{a^2}{g} \frac{\partial V}{\partial x} = 0 \quad (2)$$

Onde:

$g$  a aceleração gravitacional

$f$  é o fator de fricção do tubo

$D$  é o diâmetro do tubo

$a$  é a celeridade da onda no tubo

Os termos  $V \frac{\partial V}{\partial x}$  e  $V \frac{\partial H}{\partial x}$  descrevem o movimento do fluido de um ponto a outro e, comparados aos outros termos das equações (1) e (2), tem pouca influência no todo, podendo ser desprezados. Resultando nas novas equações simplificadas (3) e (4) a seguir:

$$g \frac{\partial H}{\partial x} + \frac{\partial V}{\partial t} + \frac{fV|V|}{2D} = 0 \quad (3)$$

$$\frac{\partial H}{\partial t} + \frac{a^2}{g} \frac{\partial V}{\partial x} = 0 \quad (4)$$

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

As equações da quantidade de movimento (3) e da continuidade (4) envolvem derivadas parciais de  $H$  e  $V$ , com o respectivo  $x$  e  $t$ , sendo ambas as variáveis funções de  $x$  e  $t$ . Reconstruindo-se as expressões envolvendo derivadas totais, e considerando a variável independente  $x$  ser uma função de  $t$ , obtém-se:

$$\frac{dH}{dt} = \left( \frac{\partial H}{\partial x} \right) \left( \frac{dx}{dt} \right) + \left( \frac{\partial H}{\partial t} \right) \quad (5)$$

$$\frac{dV}{dt} = \left( \frac{\partial V}{\partial x} \right) \left( \frac{dx}{dt} \right) + \left( \frac{\partial V}{\partial t} \right) \quad (6)$$

É possível unir as duas equações com o uso de um multiplicador  $\lambda$ , pois qualquer valor real distinto para  $\lambda$  resulta em duas equações com duas variáveis dependentes de  $H$  e  $V$  equivalentes às equações (3) e (4). Portanto, com a seleção apropriada de  $\lambda$ , obtém-se a equação simplificada (7). Expandindo esta e coletando os termos envolvendo as derivadas totais  $\frac{dH}{dt}$  e  $\frac{dV}{dt}$  segue a equação (8).

$$g \frac{\partial H}{\partial x} + \frac{\partial V}{\partial t} + \frac{fV|V|}{2D} = \lambda \left( \frac{\partial H}{\partial t} + \frac{a^2}{g} \frac{\partial V}{\partial x} \right) \quad (7)$$

$$\lambda \left( \frac{\partial H}{\partial t} + \frac{g}{\lambda} \frac{\partial H}{\partial x} \right) + \left( \frac{\partial V}{\partial t} + \frac{\lambda a^2}{g} \frac{\partial V}{\partial x} \right) + \frac{fV|V|}{2D} = 0 \quad (8)$$

Se for assumido que  $\frac{dx}{dt} = \frac{g}{\lambda}$  para  $\frac{dH}{dt}$  e  $\frac{dx}{dt} = \frac{\lambda a^2}{g}$  para  $\frac{dV}{dt}$ , tem-se (9) e (10), já que as expressões para  $\frac{dx}{dt}$  precisam ser as mesmas.

$$\frac{\lambda a^2}{g} = \frac{g}{\lambda} \quad (9)$$

$$\lambda = \pm \frac{g}{a} \quad (10)$$

Resultando em (11) abaixo:

$$\frac{dx}{dt} = \pm a \quad (11)$$

Definindo-se a constante de celeridade  $a$ , as retas com inclinação  $+a$  (referente a linha  $C+$ ) e  $-a$  (referente a linha  $C-$ ) obtém-se as seguintes equações ordinárias.

$$C+ = \left\{ \frac{g}{a} \frac{dH}{dt} + \frac{dV}{dt} + \frac{fV|V|}{2D} = 0; \frac{dx}{dt} = +a \right\} \quad (12)$$

$$C- = \left\{ -\frac{g}{a} \frac{dH}{dt} + \frac{dV}{dt} + \frac{fV|V|}{2D} = 0; \frac{dx}{dt} = -a \right\} \quad (13)$$

Utilizando-se a vazão  $Q$  do fluído que se relaciona com  $V$  por  $Q = AV$ , sendo  $A$  a área da seção transversal do tubo e considerando  $A$  constante,  $B = \frac{a}{gA}$  e  $R = \frac{f}{2gDA^2}$  as equações (12) e (13) podem ser reescritas em duas equações diferenciais ordinárias.

$$C+ = \left\{ \frac{dH}{dt} + B \frac{dQ}{dt} + aRQ|Q| = 0; dx = adt \right\} \quad (14)$$

$$C- = \left\{ \frac{dH}{dt} - B \frac{dQ}{dt} - aRQ|Q| = 0; dx = -adt \right\} \quad (15)$$

A resolução das equações ordinárias pode ser feita discretizando  $x \times t$  com  $x_i = i\Delta x$  ( $i = 0, 1, \dots, N$ ) e  $t_j = j\Delta t$  ( $j = 0, 1, \dots, N$ ), obtendo-se  $H(x_i, t_j)$  e  $Q(x_i, t_j)$ .

numericamente em cada ponto discretizado.

Integrando-se as equações (14) e (15) ao longo das linhas características resulta em:

$$\int_{H(x_{i-1}, t_{j-1})}^{H(x_i, t_j)} dH + B \int_{Q(x_{i-1}, t_{j-1})}^{Q(x_i, t_j)} dQ + R \int_{x_{i-1}}^{x_i} Q|Q|dx = 0 \quad (16)$$

$$\int_{H(x_{i+1}, t_{j-1})}^{H(x_i, t_j)} dH + B \int_{Q(x_{i+1}, t_{j-1})}^{Q(x_i, t_j)} dQ - R \int_{x_{i+1}}^{x_i} Q|Q|dx = 0 \quad (17)$$

Usando  $\Delta x = x_{i+1} - x_i$  e a aproximação:

$$\int_{x_{i+1}}^{x_i} Q|Q|dx = Q(x_i, t_j)|Q(x_{i+1}, t_{j-1})|\Delta x \quad (18)$$

E considerando  $C_p$ ,  $C_m$ ,  $B_p$  e  $B_m$  como em (19) e (20)

$$C_p = H(x_{i-1}, t_{j-1}) + BQ(x_{i-1}, t_{j-1}) \quad e \quad B_p = B + R\Delta x|Q(x_{i-1}, t_{j-1})| \quad (19)$$

$$C_m = H(x_{i+1}, t_{j-1}) + BQ(x_{i+1}, t_{j-1}) \quad e \quad B_m = B + R\Delta x|Q(x_{i+1}, t_{j-1})| \quad (20)$$

Então  $H(x_i, t_j)$  e  $Q(x_i, t_j)$  pode ser expresso como:

$$H(x_i, t_j) = C_p - B_p Q(x_i, t_j) \quad (21)$$

$$Q(x_i, t_j) = \frac{C_p - C_m}{B_p + B_m} \quad (22)$$

A cada incremento do tempo, a solução numérica do problema carrega os valores de  $H$  e  $Q$  ao longo da linha de características, como ilustrado na Figura 7. Assim, os valores de pressão ( $H$ ) e vazão ( $Q$ ) em  $t=0$ , isto é, no regime permanente inicial, são carregados no tempo

ao longo das linhas de características no interior do domínio  $x \times t$ . E as condições de contorno, nó montante (próximo ao reservatório) e nó jusante (próximo à válvula), além de completar a solução, é através destas que um regime transiente é introduzido ao sistema por meio de uma manobra hidráulica. (o fechamento da válvula no sistema simulado neste trabalho, por exemplo)

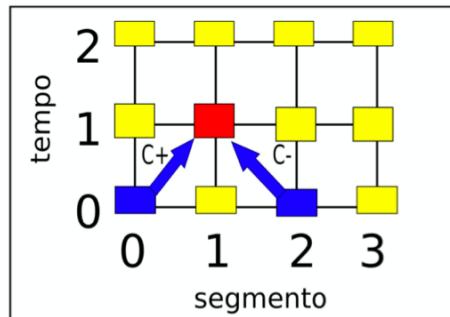


Figura 7: Representação do espaço discretizado  $x \times t$ . Em azul, as características nos pontos anteriores ( $C_+$ ) e posteriores ( $C_-$ ) no tempo anterior (tempo 0) são utilizadas para encontrar o valor no ponto vermelho no tempo posterior (tempo 1).

É essencial destacar que os valores  $H$  e  $Q$  em  $t_j$  dependem unicamente dos valores no tempo anterior, (isto é, em  $t_{j-1}$ ). Sendo assim, uma vez conhecidos os valores de  $H(x_{i-1}, t_{j-1}), H(x_{i+1}, t_{j-1}), Q(x_{i-1}, t_{j-1})$  e  $Q(x_{i+1}, t_{j-1})$ , os cálculos dos  $N$  valores de  $H(x_i, t_j)$  e  $Q(x_i, t_j)$  ao longo de  $x$  ocorrem de forma imediata.

Essa condição é fundamental para a proposta desse trabalho de aplicar estratégias de computação paralela para o problema de transiente hidráulico.

#### 1.3.4 Arquitetura Computacional e Computação Paralela.

De acordo com [17], uma definição para o termo computação paralela pode ser “uma coleção de processadores que se comunicam e cooperam entre si para resolver rapidamente grandes problemas”. Os computadores atuais são baseados na arquitetura de *Von Neumann* (Figura 8), sendo compostos por um processador, um sistema de memória e um sistema de

entrada e saída (chamada simplesmente de *I/O*) [18]. O funcionamento deste tipo de arquitetura se dá por ciclos de execução onde as instruções são buscadas pela CPU na memória, decodificadas e executadas, e cada instrução pode ou não depender de dados.

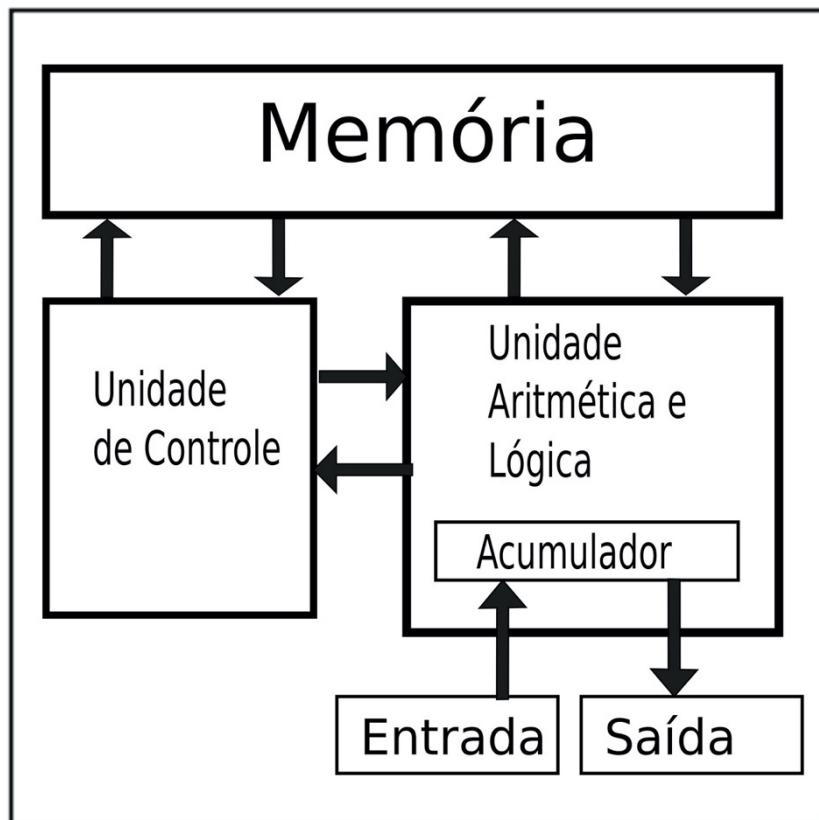


Figura 8: Arquitetura de máquinas de *Von Neumann*

Um agrupamento de máquinas de *Von Neumann*, todas trabalhando simultaneamente no mesmo objetivo computacional, pode ser considerado um computador paralelo.

Os computadores paralelos podem ser classificados pelo fluxo de instruções e de dados.

[19]

- SISD (*Single Instruction, Single Data*): arquiteturas com um único fluxo de dados e um único fluxo de instruções, como a máquina de *Von Neumann*.
- SIMD (*Single Instruction, Multiple Data*): arquiteturas com uma única instrução, mas com paralelismo nos dados, como o CUDA.

- MISD (*Multiple Instruction, Single Data*): arquiteturas com múltiplas instruções funcionando ao mesmo tempo em um mesmo conjunto de dados
- MIMD (*Multiple Instruction, Multiple Data*): arquiteturas com Múltiplas instruções funcionando ao mesmo tempo sobre um múltiplo conjunto de dados, como cluster do tipo Beowulf ou computadores massivamente paralelos.

Tanembaum [18] apresenta dois padrões de sistema de memória:

- Memória compartilhada: onde a memória do sistema é diretamente acessível por todos os processadores Figura 9.
- Memória distribuída: onde cada processador tem sua própria memória Figura 10.

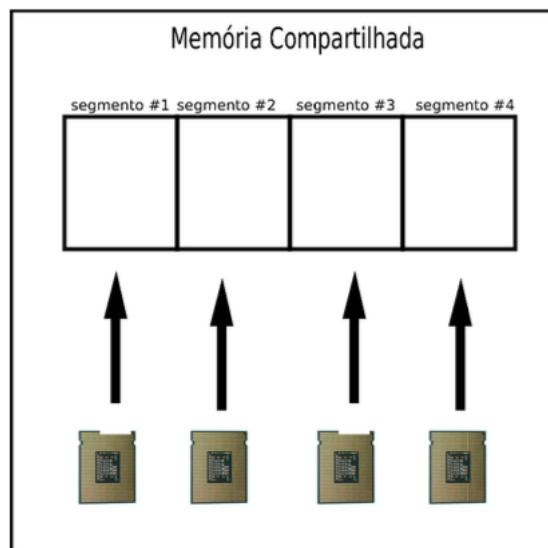


Figura 9: Visão geral de um sistema de memória

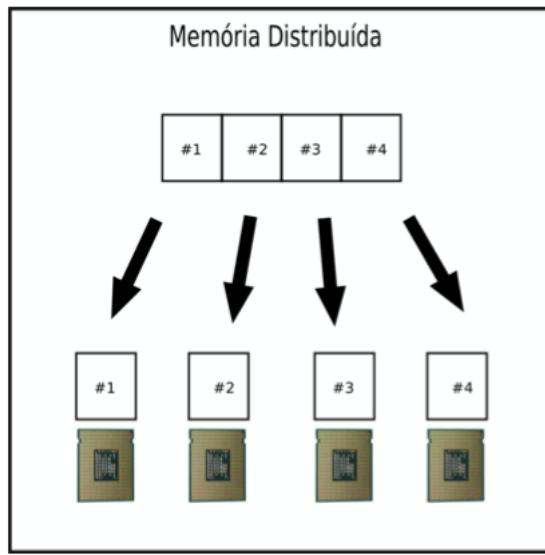


Figura 10: Visão geral de um sistema de memória compartilhada

Tanembaum [18] ainda classifica as técnicas de computação paralela conforme a unidade de processamento:

- Computadores fracamente acoplados: possuem baixa largura de banda, alta taxas de atraso para troca de informações, podendo ou não estar fisicamente no mesmo computador. Computadores que utilizam o padrão MPI (*Message Passing Interface*) são exemplos dessa classificação [20].
- Computadores fortemente acoplados: possuem alta largura de banda com baixo atraso para troca de mensagens entre as unidades de processamento. A arquitetura GPU e a API OpenMP são exemplos dessa classificação.

### 1.3.5 Arquitetura GPU / CUDA

A arquitetura GPU resume-se a um computador com uma ou mais CPUs (*host*) e uma ou mais placas de vídeo com processadores paralelos (Figura 11) com várias unidades aritméticas de execução (*devices*). O *device* atua como um co-processador que executa em paralelo com o *host*.

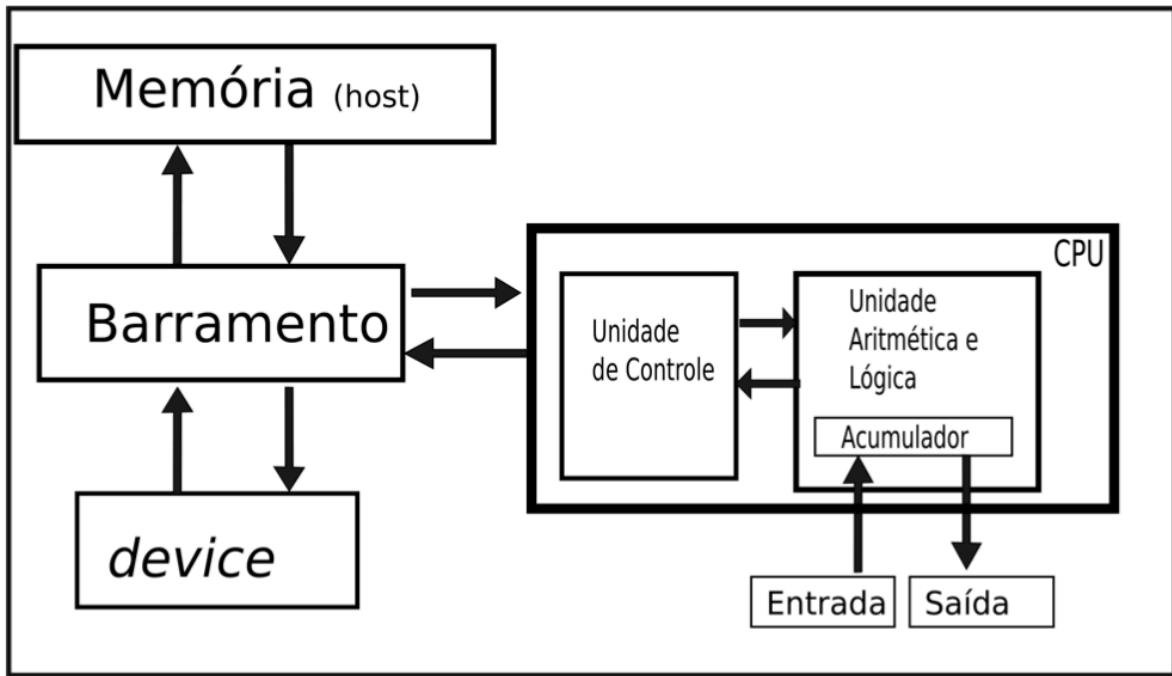


Figura 11: Visão geral da arquitetura GPU.

Dentre os fornecedores de GPU, a Nvidia foi a pioneira no uso de processamento gráfico para computação de propósito geral com a criação da arquitetura CUDA. [21]

A Nvidia construiu as unidades lógicas de processamento (ULA) de suas placas de vídeo de forma que cumprissem os requisitos do IEEE para aritmética de ponto flutuante de precisão, projetadas para usar um conjunto de instruções feito sob medida para computação geral, em vez de especificamente para gráficos.

Além disso, as unidades de execução na GPU passaram a ter acesso de leitura e gravação na memória, bem como acesso a um cache gerenciado por software conhecido como memória compartilhada. A Nvidia também criou um conjunto de ferramentas para escrever código para a GPU, e forneceu um *driver* de *hardware* especializado para explorar todo poder computacional da arquitetura CUDA.

Esse conjunto de ferramentas inclui, além de um ambiente de software, suporte a diversas linguagens como Fortran, OpenCL, openACC, C e outras. A linguagem escolhida neste trabalho foi a linguagem CUDA-C, pois permite aos programadores que usem uma

linguagem de alto nível padrão C-ANSI estendido, sendo que a linguagem C foi usada para o desenvolvimento de outras atividades durante o curso de graduação.

Todos esses recursos da arquitetura CUDA foram adicionados para criar uma GPU que se destacaria em computação, além de ter um bom desempenho em tarefas gráficas tradicionais.

Há uma ação importante a se realizar quando se escolhe trabalhar com a arquitetura CUDA, ela é verificar se o *device* (GPU) é compatível e qual a sua capacidade computacional (*capability*), definido por um número de revisão principal e secundário.

O *capability* vai indicar as limitações e características de uma certa versão. Números de revisão principal iguais assinalam uma mesma arquitetura de núcleo, que podem incluir novos atributos que não são suportados por arquiteturas de núcleo anteriores. A arquitetura CUDA se apresenta promissora para algoritmos do tipo SIMD, devido a sua organização interna e hierarquia de alguns componentes.

Na arquitetura CUDA, um kernel pode ser definido como uma rotina especial que será executada no GPU (*device*) [22], diferentemente de outras rotinas que executam na CPU.

Para iniciar um código paralelo na GPU é preciso instruir o sistema de tempo de execução CUDA sobre quantas cópias paralelas do kernel iniciar. Essas cópias paralelas são chamadas blocos.

O *runtime* tempo de execução CUDA permite que esses blocos sejam divididos em threads 1D, 2D ou 3D, sendo que cada bloco possui uma organização 2D, dimensionado em 5 x 3 x 1, totalizando 15 *threads* por bloco. [23][22]. Os blocos são organizados em *grids* 1D ou 2D. Na Figura 12, cada *grid* possui uma organização 2D, dimensionada em 3 por 2 blocos, totalizando 6 blocos na grid do primeiro *kernel* [22].

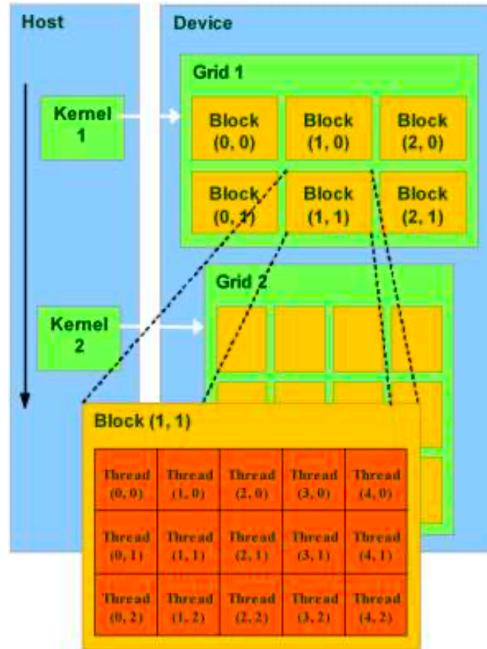


Figura 12: Exemplo de *thread*, bloco, *grid* e *kernel*. [16]

Existe uma relação entre custo, desempenho e preço de memória. Quanto mais rápido o acesso a memória, maior será o seu custo e menor o seu tamanho. [25] Este conceito se aplica aos tipos de memória existentes no *device* (Figura 13).

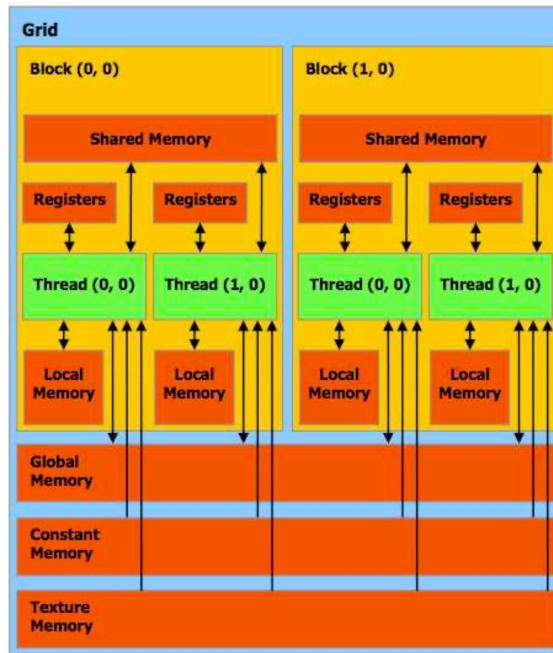


Figura 13: Hierarquia de memórias em CUDA. [16]

Nas memórias dos *devices* com *capability* até 1.x precisam de sequenciamento e alinhamento a cada bloco de 16 bytes para a leitura e gravação por *grid*, sendo lenta e não cacheável (*uncached*). Nos *devices* com *capability* 2.x, a memória global é alocada no *device* e acessada através de transações de 32, 64 ou 128 bytes, ainda considerada lenta, mas cacheável. Essa memória é acessível por todas as linhas de execução do *grid*, e é possível transferir dados do host para o *device* e vice e versa.

Com centenas de ULAs na GPU, muitas vezes o gargalo não é a taxa de transferência aritmética do chip, mas sim a largura de banda de memória do chip. A linguagem CUDA C disponibiliza outro tipo de memória conhecida como memória constante (*constant memory*), que, como o nome pode indicar, é utilizada para armazenar dados que não serão alterados durante a execução do kernel. O *hardware* Nvidia fornece 64 KB de memória constante, que, em algumas situações, seu uso em vez de memória global pode reduzir a largura de banda da memória necessária.

Há ainda outro tipo de memória somente leitura que está disponível para uso em programas escritos em CUDA C, a memória de textura (*texture memory*). Como a memória constante, a memória de textura é armazenada em cache no chip, portanto, em algumas situações, ela fornecerá maior largura de banda efetiva, reduzindo as solicitações de memória para DRAM fora do chip.

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, rápidas e pequenas. Já a memória local (*local memory*) é lenta e não cacheável e é utilizado para todas as situações de propósito de uso local que não se extrapolam o limite da memória registradores.

A GPU foi projetada para executar milhares de linhas de execução em paralelo. Assim programas com diversas operações aritméticas (mas altamente paralelizáveis) podem se beneficiar com o uso da GPU em detrimento da CPU, que foi projetada para executar, à velocidade máxima, uma única linha de execução, com instruções sequenciais. [26]

### **1.3.6 OpenMP e por que usá-lo**

*OpenMP* é uma *API* baseada no modelo de programação paralela de memória compartilhada para arquiteturas de múltiplos processadores e seu código utiliza diretivas de compilação, a biblioteca de execução e 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 existentes no hardware. Com isso o programa utiliza todos os processadores disponíveis. O computador que executar o código *OpenMP* precisa ter a biblioteca *OpenMP* previamente instalada.

A biblioteca OpenMP encapsulou muitas funções que abstrai do programador algumas complexidades do paralelismo, assim a curva de aprendizado do *OpenMP* mostrou-se menor comparado a GPU, e por isso esta *API* foi escolhida para este trabalho.

### **1.3.7 Métricas para análise de desempenho**

Considerando o objetivo deste trabalho, é essencial que sejam utilizadas métricas para quantificar o desempenho do software nos diferentes hardwares, a fim de se obter uma

comparação. Alguns exemplos de métricas de *software* são número de linhas de código, tempo para realizar a programação e custo para realização de uma tarefa, podendo essas métricas ter enfoque no desempenho do produto, do processo ou do projeto. [27]

Uma falácia que deve ser evitada nas métricas de software é realizar a metrificação baseada em opiniões subjetivas e enviesadas. É necessário se apoiar em dados e estatísticas oriundos de valores quantitativos e matemáticos.

Para este trabalho, foram utilizadas as métricas de desempenho de produto *speedup* e eficiência computacional.

O *Speedup* refere-se ao aumento de velocidade observado quando se executa um determinado processo em p processadores em relação à execução deste processo em 1 processador. Então, tem-se a equação (23).

$$S_p = \frac{T_1}{T_p} \quad (23)$$

Onde

$T_1$  é o tempo de execução em um único processador

$T_p$  é o tempo de execução em p processadores

## 2 DESENVOLVIMENTO

### 2.1 Pseudocódigo

Um pseudocódigo que implementa o algoritmo do método das características é apresentado na Figura 14. Esta solução foi implementada em um código escrito em C que, através de parâmetros, resolve o problema de transitório hidráulico de forma serial ou de forma paralela, com o uso da API OpenMP ou com o uso da arquitetura CUDA.

1	Ler parâmetros na linha de comando
2	De acordo com os parâmetros definir qual versão da função de resolução do problema será chamada
3	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)
4	Cálculo das constantes $dx = 1/n$ $dt = dx/a$
5	Alocação dinâmica das matrizes H (dx,dt) e Q (dx,dt) Copiar matrizes para o device
6	Cálculo do Regime Permanente Inicial Para $x=0, x \leq n$ , X++ { calcular H[x] calcular Q[x] }
7	Enquanto $t \leq t_{\text{max}}$ { t = t + dt Para $x=0, x \leq n$ , X++ { Calcular CP, BP, CM, BM (baseado no dt anterior) $Q[x] = (CP - CM) / (BP + BM)$ $H[x] = CP - BP * Q[x]$ } }
8	Gravar resultados caso defino na linha de comando

Figura 14: Pseudocódigo que implementa o método das características

O problema do transitório é modelado segundo um arquivo de texto e informado para o programa via parâmetros de entrada.

Os parâmetros no arquivo do problema são altura do reservatório (hr), fator de fricção Darcy-Weisbach (f) (um parâmetro adimensional que é utilizado para calcular a perda de carga em uma tubulação devida ao atrito), largura do tubo (l), diâmetro do tubo (d), número de segmentos do tubo (n), tempo máximo de simulação (tmax), celeridade da onda (a) (trata da velocidade com que uma onda de pressão se propaga num fluido, dado em metros por segundo) e o coeficiente de descarga multiplicado pela área da válvula (cdao), informados nessa ordem, um em cada linha. 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.

Foram modelados dois dutos reais, com dados obtidos no site da CETESB, e um duto fictício, com parâmetros conforme apresentados na Tabela 1, sendo que os valores de altura do reservatório (hr), fator de fricção(f), a celeridade da onda (a) (referente ao cálculo de um cano de metal e água [28]) e o coeficiente de descarga (cdao) são iguais para os três problemas.

Tabela 1: Parâmetros dos problemas simulados

Parâmetros	Problema Fictício	Duto OSPLAN II	Duto OPASA 10pol
hr	150	150	150
f	0.035	0.035	0.035
l	120000	154000	98800
d	0.3	0.4572	0.254
a	1030	1030	1030
cdao	0.0009	0.0009	0.0009

## 2.2 Implementação

O pseudocódigo da Figura 14 é uma solução do problema do transiente hidráulico resolvido pelo método das características. Esse pseudocódigo foi implementado na linguagem C, levando-se em consideração as três arquiteturas expostas neste trabalho: processamento serial, *multicore* e CUDA.

Ao se executar a o programa, deve-se informar os parâmetros que definem a escolha da solução-arquitetura, o arquivo com os parâmetros do problema, mensagens de debug, mensagens que poderão ser exibidas durante a execução, arquivo de saída com a solução, número de repetições e outros, conforme a Tabela 2 abaixo:

Tabela 2: Parâmetros que devem ser informados para execução do programa

-h	Exibe mensagem de ajuda
-v	Exibição de todas as mensagens do programa durante a execução
-e <file>	Arquivo de entrada.
-s <file>	Arquivo de saída com o resultado.

-m <modo>	0 = Singlecore   1 = Multicore   2 = CUDA: Utiliza a versão indicada.
-t <num>	Número de Threads.
-d <num>	Número de vezes que a simulação será executada (Simula o tamanho do problema).
-p	Roda as instâncias do problema de forma paralela.
-k	Exibe os tempos apenas

## 2.3 Desenvolvimento Serial

Após a leitura dos parâmetros de entrada, ocorre o processamento do arquivo do problema para obter o tamanho do segmento a ser calculado (dx) e o intervalo entre um tempo e outro (dt).

No passo 5, uma matriz dinâmica é criada contendo a estrutura dos dados para pressão (H), vazão (Q) e tempo (t), pois as variáveis celeridade da onda (a), número de segmentos (n) e tamanho do tubo (l) possuem impacto direto no tamanho das malhas computacionais.

Neste ciclo, uma única matriz de uma estrutura contendo os três números de ponto flutuante (*float\**) é declarada e alocada dinamicamente, conforme apresentado na Figura 15 abaixo.

```

struct _HQT {
    float H;
    float Q;
    float T;
};

struct _HQT *matrizHQT;

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

matrizHQT = (struct _HQT *)malloc(sizeof(struct _HQT) * sizeMATRIX);

```

Figura 15: Trecho de código do passo 5 em desenvolvimento serial

A escolha de se criar uma *struct* com as três variáveis melhora o desempenho, pois tira proveito da arquitetura de alocação de memória e das estratégias de cache da arquitetura Intel, agrupando os dados que serão processados na memória cache. Além de simplificar o desenvolvimento, uma vez que elimina laços desnecessários e mantém um único vetor e uma variável de índice (ns) para cada variação em  $\Delta t$ .

A variável de índice (ns) representa o número de segmentos, somado do nó de montante e o nó de jusante, isto é  $ns = n + 2$ .

No passo 6 ocorre o cálculo das características no tempo inicial (Regime Permanente Inicial) em todos os segmentos do tubo no início da simulação. Na estrutura desenvolvida no passo 6, o cálculo é realizado em um laço de repetição *for* e os valores são atualizados na struct do vetor matrizHQT, conforme destacado no trecho da Figura 16.

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

for(i = 0; i < ns; i++) {
    atualHQ[i].H = hr - (i) * r * qo * qo;
    atualHQ[i].Q = qo;
    matrizHQT[i].H = atualHQ[i].H;
    matrizHQT[i].Q = atualHQ[i].Q;
    matrizHQT[i].T = t;
}
```

Figura 16: Trecho de código do passo 6 no desenvolvimento serial

No passo 7 ocorre o cálculo da pressão e da vazão baseadas nas características no tempo anterior e nos segmentos vizinhos (Regime Transiente). A Figura 7 anterior ilustra essa dependência dos cálculos para o Regime Transiente.

O histórico de valores simulados esperado é alcançado após o cálculo de todos os elementos de pressão (H) e vazão (Q) no tempo (T). Então uma coluna de um determinado segmento representa o histórico do transiente desse mesmo segmento.

A etapa do passo 7, apresentado abaixo, é uma implementação serial com uma única instrução, o cálculo de Cp, Cm, Bp e Bm em diferentes pontos das matrizes pressão e vazão. Por isso, esse passo é o mais adequado a ser implementado com o uso de arquiteturas paralelas SIMD.

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

    //Pontos interiores;
    for(i = 1; i < ns; i++) {
        cp = atualHQ[i-1].H + b * atualHQ[i-1].Q;
        bp = b + r * fabs(atualHQ[i-1].Q);
        cm = atualHQ[i+1].H - b * atualHQ[i+1].Q;
        bm = b + r * fabs(atualHQ[i+1].Q);
        proximolHQ[i].Q = (cp - cm) / (bp + bm);
        proximolHQ[i].H = cp - bp * proximolHQ[i].Q;
    }

    //Condição de contorno de montante: Reservatório

    proximolHQ[0].H = hr;
    cm = atualHQ[1].H - b * atualHQ[1].Q;
    bm = b + r * fabs(atualHQ[1].Q);
    proximolHQ[0].Q = (hr - cm)/bm;

    //condição de contorno de jusante: Válvula
```

```

cp = atualHQ[ns-2].H + b * atualHQ[ns-2].Q;
proximolHQ[ns-1].Q = 0;
proximolHQ[ns-1].H = cp;

//Atualização

for (i = 0; i < ns; i++) {
    atualHQ[i].H = proximolHQ[i].H;
    atualHQ[i].Q = proximolHQ[i].Q;

    unsigned int ind = (controle * ns) + i;
    if (ind < sizeMATRIX) {
        matrizHQT[ind].H = atualHQ[i].H;
        matrizHQT[ind].Q = atualHQ[i].Q;
        matrizHQT[ind].T = t;
    }
}

}

```

Figura 17: Trecho de código do passo 7 no desenvolvimento serial

A Figura 18 ilustra a evolução computacional que ocorre no passo 7 em um exemplo de uma malha computacional com dez segmentos e oito tempos ( $\Delta t_s$ ). Na Figura 18(a), o segmento 3 no tempo 1, em azul, é calculado, utilizando-se a informação no tempo anterior nos segmentos 2 (C+) e 4 (C-), em vermelhos. Os cálculos ocorrem de forma análoga em Figura 18(b) para o segmento 4 no tempo 1, em Figura 18(c) para o segmento 5 no tempo 1 e em Figura 18(d) para o segmento 6 no tempo 1 é calculado.

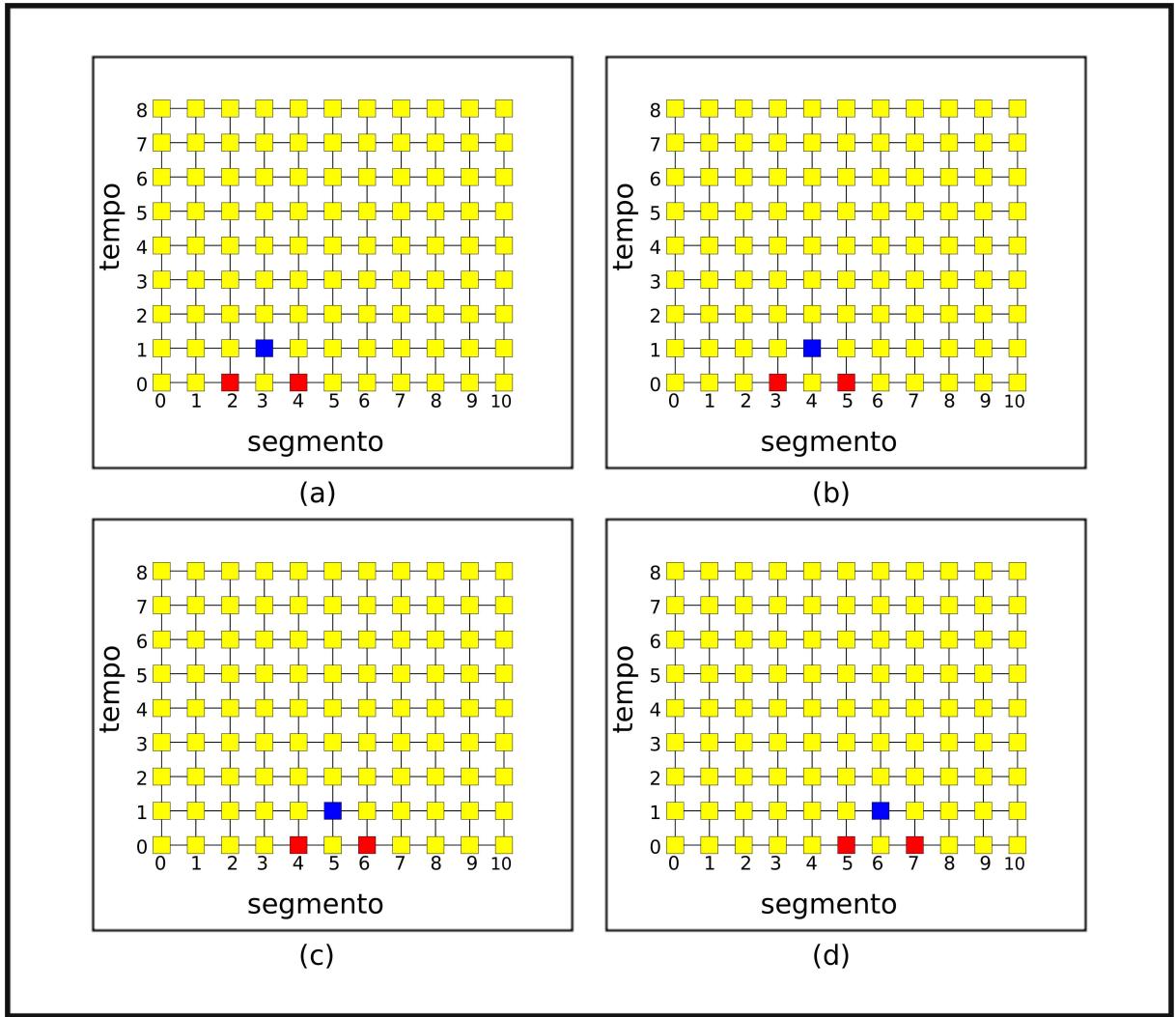


Figura 18: Evolução computacional do passo 7, em processamento serial, onde o cálculo de pressão e vazão de um segmento (em azul) depende do valor de segmentos em tempo anterior (em vermelho).

## 2.4 Desenvolvimento Paralelo

Como declarado anteriormente, o tempo necessário para o aprendizado da API OpenMP foi menor quando comparado ao tempo necessário para o aprendizado da arquitetura CUDA, já que a API encapsulou boa parte das dificuldades encontradas em desenvolvimento paralelo, entre elas, a sincronização.

Como explicado anteriormente, o passo 6 realiza o cálculo Regime Permanente Inicial e o passo 7, o cálculo do Regime Transiente, utilizando os valores das matrizes de pressão e

vazão (struct do vetor matrizHQT) em laços. Por essas características de implementação, esses passos foram escolhidos para serem paralelizados.

O trecho de código referente ao passo 6 pode ser observado abaixo na Figura 19. 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(atualHQ,matrizHQT,hr,r,qo,t,ns,chunk)
private(i)
{
    #pragma omp for schedule(dynamic,chunk) nowait
    for(i = 0; i < ns; i++) {
        atualHQ[i].H = hr - (i) * r * qo * qo;
        atualHQ[i].Q = qo;
        matrizHQT[i].H = atualHQ[i].H;
        matrizHQT[i].Q = atualHQ[i].Q;
        matrizHQT[i].T = t;
    }
}
```

Figura 19: Trecho de código do passo 6 no desenvolvimento paralelo

No passo 7 pode-se ver os trechos paralelizados onde ocorre o cálculo dos segmentos internos, atualização das variáveis e atualização das matrizes na Figura 20 em seguida.

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

    #pragma omp parallel shared(atualHQ,proximolHQ,r,b,ns,chunk)
private(i,cm,bm,cp,bp)
```

```

{

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

    //Condição de contorno de montante: Reservatório

    proximolHQ[0].H = hr;
    cm = atualHQ[1].H - b * atualHQ[1].Q;
    bm = b + r * fabs(atualHQ[1].Q);
    proximolHQ[0].Q = (hr - cm)/bm;

    //condição de contorno de jusante: Válvula

    cp = atualHQ[ns-2].H + b * atualHQ[ns-2].Q;
    proximolHQ[ns-1].Q = 0;
    proximolHQ[ns-1].H = cp;
    //Atualização

    #pragma omp parallel shared(atualHQ,proximolHQ,matrizHQT,ns,t,chunk)
private(i)

{
    #pragma omp for schedule(dynamic,chunk) nowait
        for (i = 0; i < ns; i++) {
            atualHQ[i].H = proximolHQ[i].H;
            atualHQ[i].Q = proximolHQ[i].Q;
            unsigned int ind = (controle * ns) + i;
            if (ind < sizeMATRIX) {
                matrizHQT[ind].H = atualHQ[i].H;
            }
        }
}

```

```

        matrizHQT[ind].Q = atualHQ[i].Q;
        matrizHQT[ind].T = t;
    }
}
}
}

```

Figura 20: Trecho de código do passo 7 no desenvolvimento paralelo

A Figura 21 ilustra a evolução computacional de duas *threads*, *thread 1* (vermelha) e *thread 2* (verde), no passo 7 já paralelizado com a API *OpenMP*. Em Figura 21(a), *thread 1* calcula o segmento 1 no tempo 1, em vermelho, (utilizando os segmentos 0 e 2, em rosa claro, no tempo 0, C+ e C- respectivamente), enquanto a *thread 2* realiza o mesmo cálculo no segmento 2, em verde claro, (utilizando os segmentos 1 e 3, em verde escuro, no tempo 0, C+ e C- respectivamente). É possível observar a evolução dos cálculos na sequência de figuras, com o mesmo padrão de cores. Nelas, de forma análoga, observa-se que são calculados os segmentos 3 e 4 em Figura 21(b), segmentos 5 e 6 em Figura 21(c) e segmentos 7 e 8 em Figura 21(d). Esse processo ocorre até o cálculo completo de todos os elementos da Pressão (H) e Vazão (Q) do sistema.

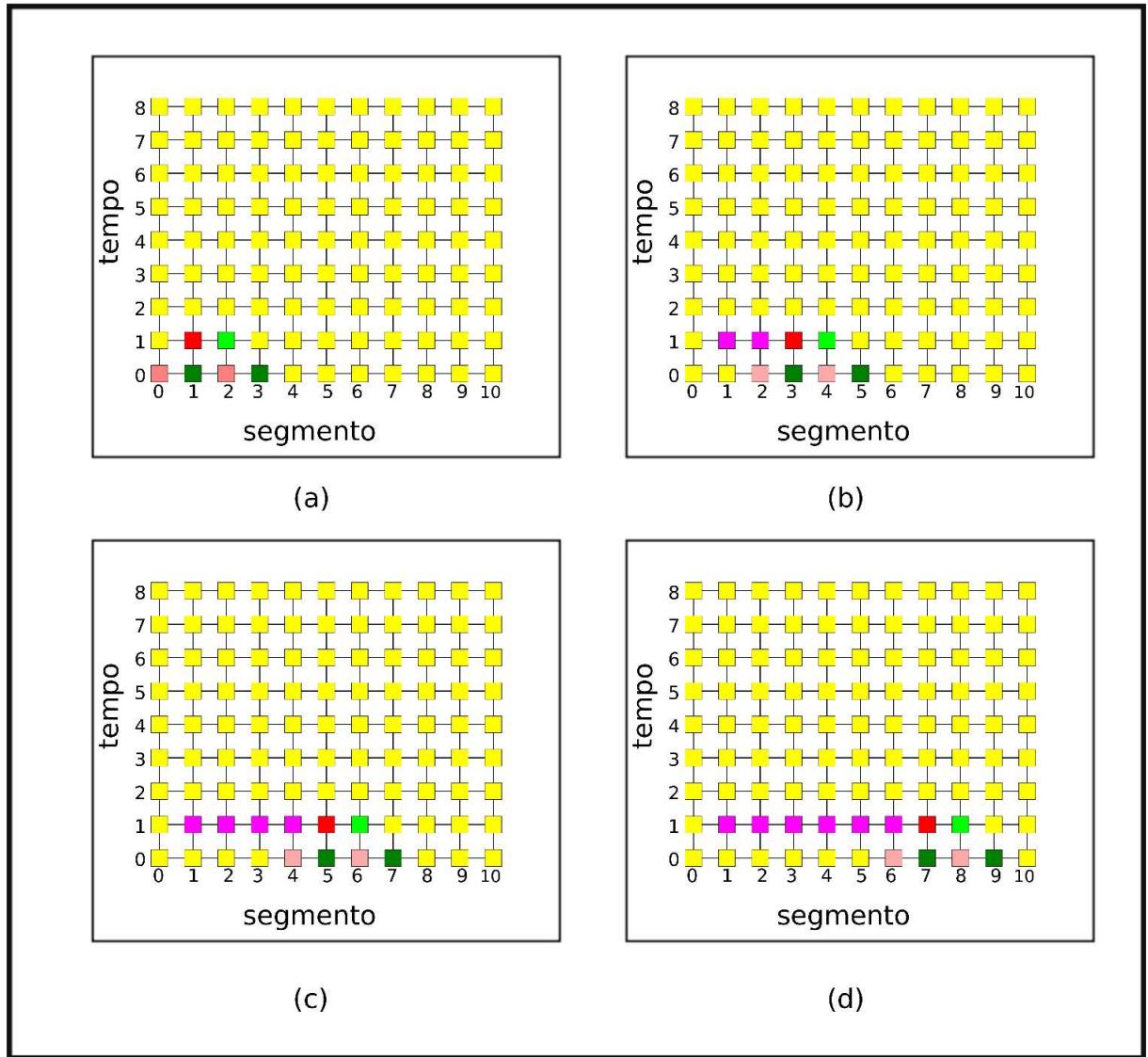


Figura 21: Evolução computacional do passo 7, em processamento *multicore*, onde ocorre o cálculo de dois segmentos simultaneamente.

## 2.5 Desenvolvimento CUDA

Na arquitetura CUDA, a implementação paralela realiza tanto alocações dinâmicas na memória RAM (`malloc`) e quanto na memória da placa de vídeo, ou *device*, (`cudaMalloc`). O trecho seguinte de código (Figura 22) mostra que os cálculos processados na placa de vídeo ficam armazenados na memória global da placa até serem transferidos para memória RAM, com o comando `cudaMemcpy`.

```

struct _HQT {
    float H;
    float Q;
    float T;
};

struct _HQT *matrizHQT;

size_t sizeMATRIX = N; /* sizeof(float);
matrizHQT = (_HQT *)malloc(sizeof(_HQT) * sizeMATRIX);

// Inicializa H e Q
initZeroHQT(matrizHQT , sizeMATRIX);
//Aloca vetores na memória do device
HANDLE_ERROR( cudaMallocManaged(&_dMatrizHQT, (sizeof(_HQT) *
sizeMATRIX)) );

//Dados : Host -> Device
HANDLE_ERROR( cudaMemcpy(_dMatrizHQT, matrizHQT, (sizeof(_HQT) *
sizeMATRIX), cudaMemcpyHostToDevice) );
...
calc_HQ_RegimeTransiente<<<nblocks, nthreads,
nthreads*sizeof(_HQ)>>>(_dMatrizHQT);
...
//Dados : Device -> Host
HANDLE_ERROR( cudaMemcpy(matrizHQT, _dMatrizHQT, (sizeof(_HQT) *
sizeMATRIX), cudaMemcpyDeviceToHost) );...

```

Figura 22: Trecho de código que exibe a utilização de malloc, cudaMalloc e cudaMemcpy

Para [16], o desenvolvimento com uso da arquitetura CUDA pode apresentar um problema que merece atenção. Ele é a possibilidade de uma *thread* concluir a execução mais rápida que outra, e solicitar uma informação que ainda não tenha sido calculada pela outra *thread*. Isso foi resolvido usando um ponto de sincronia, caso contrário os resultados obtidos não seriam confiáveis.

Outro problema, observado empiricamente durante o desenvolvimento desse trabalho, é o tamanho da malha computacional alocada dinamicamente, que não pode exceder o tamanho da memória da placa de vídeo. Considerando que esta pode ser notadamente menor que a memória de uso geral do computador, este é um ponto que mereceu ser tratado com bastante prudência.

A Figura 23 ilustra o progresso dos cálculos realizados utilizando as *threads* de uma arquitetura CUDA. Na Figura 23(a), as *threads* calculam os segmentos ímpares no tempo 1 (em rosa), utilizando valores no tempo 0 (em verde) e posteriormente, na Figura 23(b) os segmentos pares também no tempo 1. De forma análoga, na Figura 23(c) ocorrem os cálculos dos segmentos ímpares no tempo 2 e em seguida, na Figura 23(d), dos segmentos pares.

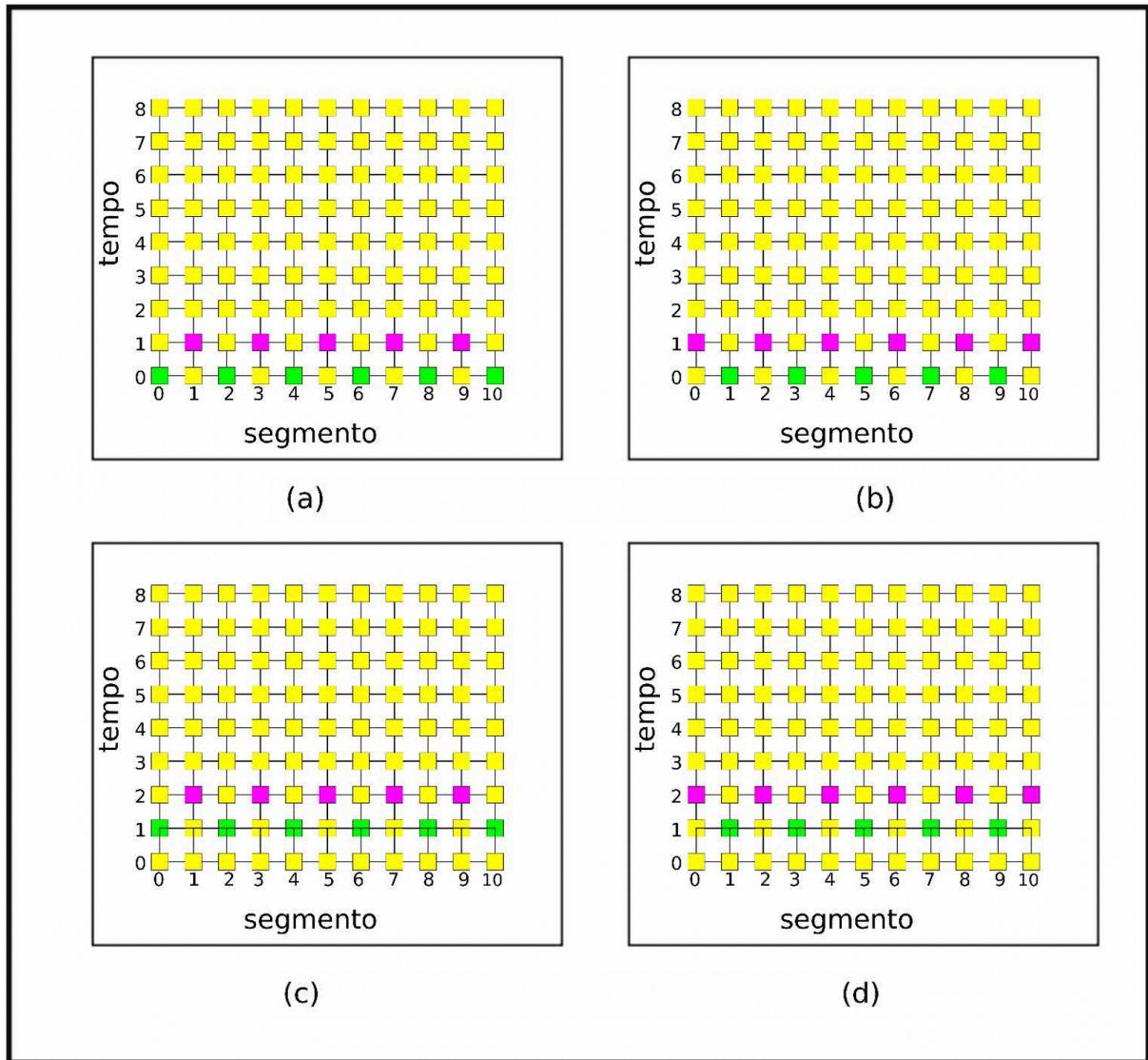


Figura 23: Evolução do cálculo executado no passo 7, implementado em CUDA, onde o cálculo de pressão e vazão ocorre em todos segmentos ímpares (rosa) de um tempo simultaneamente, e em seguida são calculados todos os segmentos pares (verde) do mesmo tempo.

Como explicado anteriormente, foi imprescindível definir um ponto de sincronia, ou *syncthread*, onde todas as *threads* da GPU devem aguardar a execução uma das outras, a fim de evitar perda de confiabilidade nos dados calculados. O trecho do código a seguir (Figura 24) exibe que o ponto de sincronia escolhido foi na fronteira entre um tempo para o outro seguinte, antes que os valores calculados na matrizes pressão, vazão e tempo (MatrizHQT) fossem atualizados.

```

__global__ void calc_HQ_RegimeTransiente(_HQT *_dMatrizHQT) {
    ...
    //Atualização
    tempoAtual[indice].H = tempoProximo[indice].H;
    tempoAtual[indice].Q = tempoProximo[indice].Q;
    __syncthreads();
    _dMatrizHQT[(_controle * _ns) + _i].H = tempoAtual[_i].H;
    _dMatrizHQT[(_controle * _ns) + _i].Q = tempoAtual[_i].Q;
    _dMatrizHQT[(_controle * _ns) + _i].T = _t;
    ...
}

```

Figura 24: Trecho de código do passo 7 na implementação CUDA

O diagrama da Figura 25 ilustra o funcionamento do ponto de sincronia durante a execução da simulação. Em destaque, na Figura 25(a) está o intervalo da matriz que será trabalhado pelo bloco, e ao lado estão os pontos que serão calculados até a próxima sincronização. Na Figura 25(b), observa-se a evolução dos cálculos realizados por duas *threads*, calculando os segmentos 1 e 2 (em t0), os segmentos 3 e 4 (em t1) e finalmente, o segmento 0 (em t2). Note que na Figura 25(c) ocorre o ponto de sincronia, ou seja, os valores da linha no tempo 2 serão calculados apenas quando todos os valores dos segmentos da linha no tempo 1 estiverem finalizados.

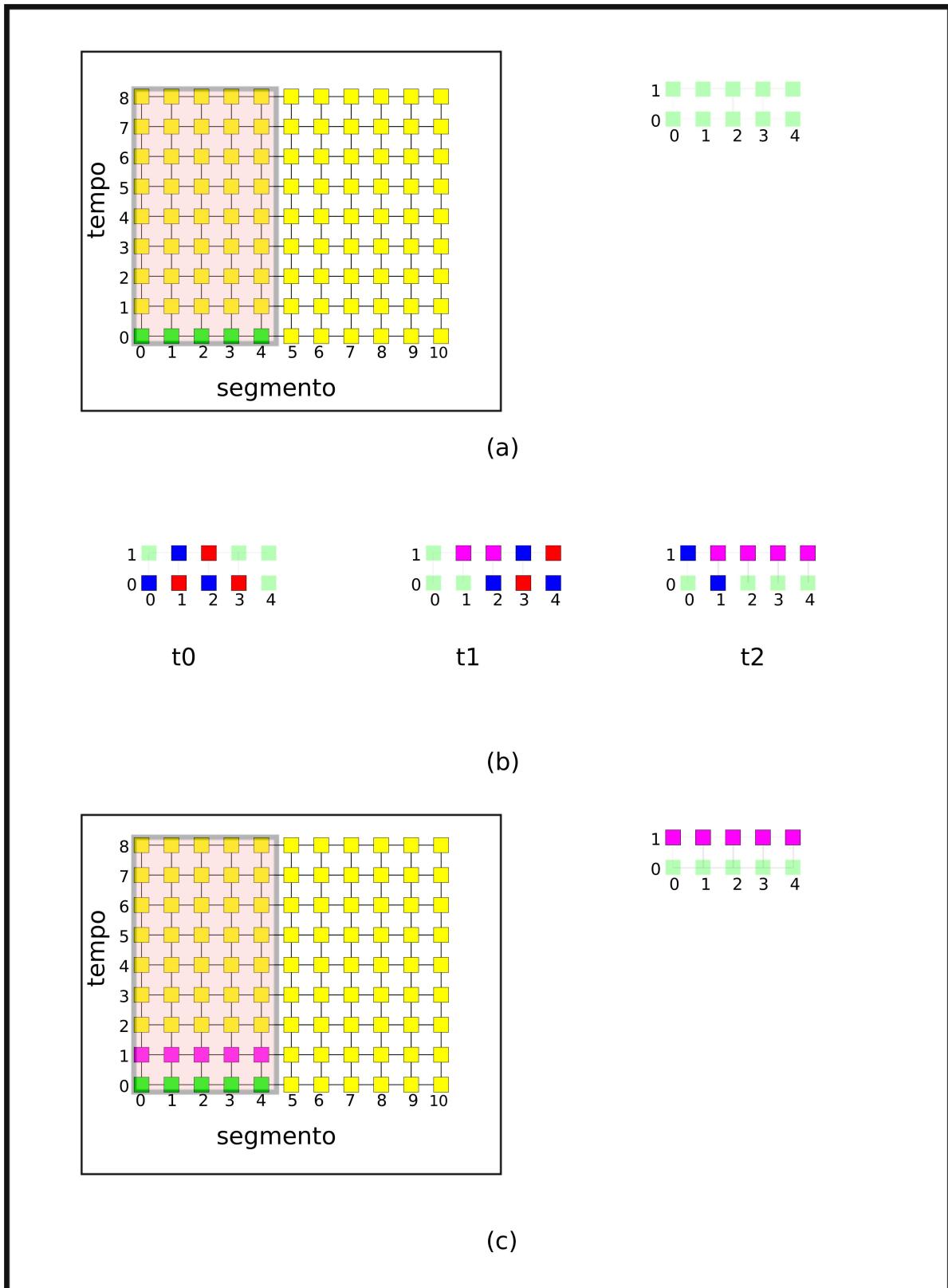


Figura 25: Funcionamento do ponto de sincronização. Em (a), em destaque, os segmentos que serão trabalhados até a próxima sincronização. Em (b), a evolução do processamento de duas threads para o cálculo dos segmentos 1 e 2 (t0), 3 e 4 (t1) e 0 (t1). Em (c) ocorre o ponto de sincronização e o processamento só continua quando todos os segmentos da linha 1, daquele bloco, foram calculados.

Como visto na ilustração anterior, existe um intervalo de tempo em que o programa verifica se todos os cálculos foram efetivamente realizados antes de prosseguir com o processamento. Apesar da utilização de *syncthreads* consumir tempo de processamento e, portanto, causar uma perda de desempenho durante a simulação, isto é indispensável para que os resultados obtidos sejam confiáveis.

O trecho de código abaixo mostra o método utilizado para encontrar o número de blocos que devem ser invocados. A quantidade de *threads* por bloco é definida por parâmetro na linha de comando, e o melhor valor é definido de forma empírica, testando-se os valores e utilizando o número que obteve o menor tempo de processamento.

```

int main(int argc, char** argv) {
    ...
    while ( (c = getopt (argc, argv, "e:s:m:c:t:d:pvhk")) != -1) {
        switch(c) {
            ...
            case 't':
                nthreads = atoi(optarg);
                chunk = nthreads;
                break;
            ...
        }
    }
    ns = n + 2; // Numero de segmentos + segmento de montante + segmento de jusante
    nblocks = (ns + nthreads) / nthreads;...

    calc_HQ_RegimeTransiente<<<nblocks, nthreads,
    nthreads*sizeof(_HQ)>>>(_dMatrizHQT);
    ...
}

```

Figura 26: Trecho de código que encontra o número de blocos a ser utilizado

A Figura 27, ilustra a sequência de cálculos que são realizados em blocos na arquitetura CUDA, usando a sincronização, enfatizando a maior velocidade de processamento nessa

arquitetura. Primeiro, o bloco é selecionado (Figura 27 a), e em seguida é realizado cálculo de todos os segmentos pertencentes a esse bloco (Figura 27 b). O bloco seguinte a ser trabalhado é selecionado de forma a conter a última coluna, com valores já calculados, do bloco anterior (Figura 27 c). Nesse novo bloco são executados os cálculos de todos os segmentos (Figura 27 d), para então, de forma cíclica, ocorrer a escolha do próximo bloco a ser processado.

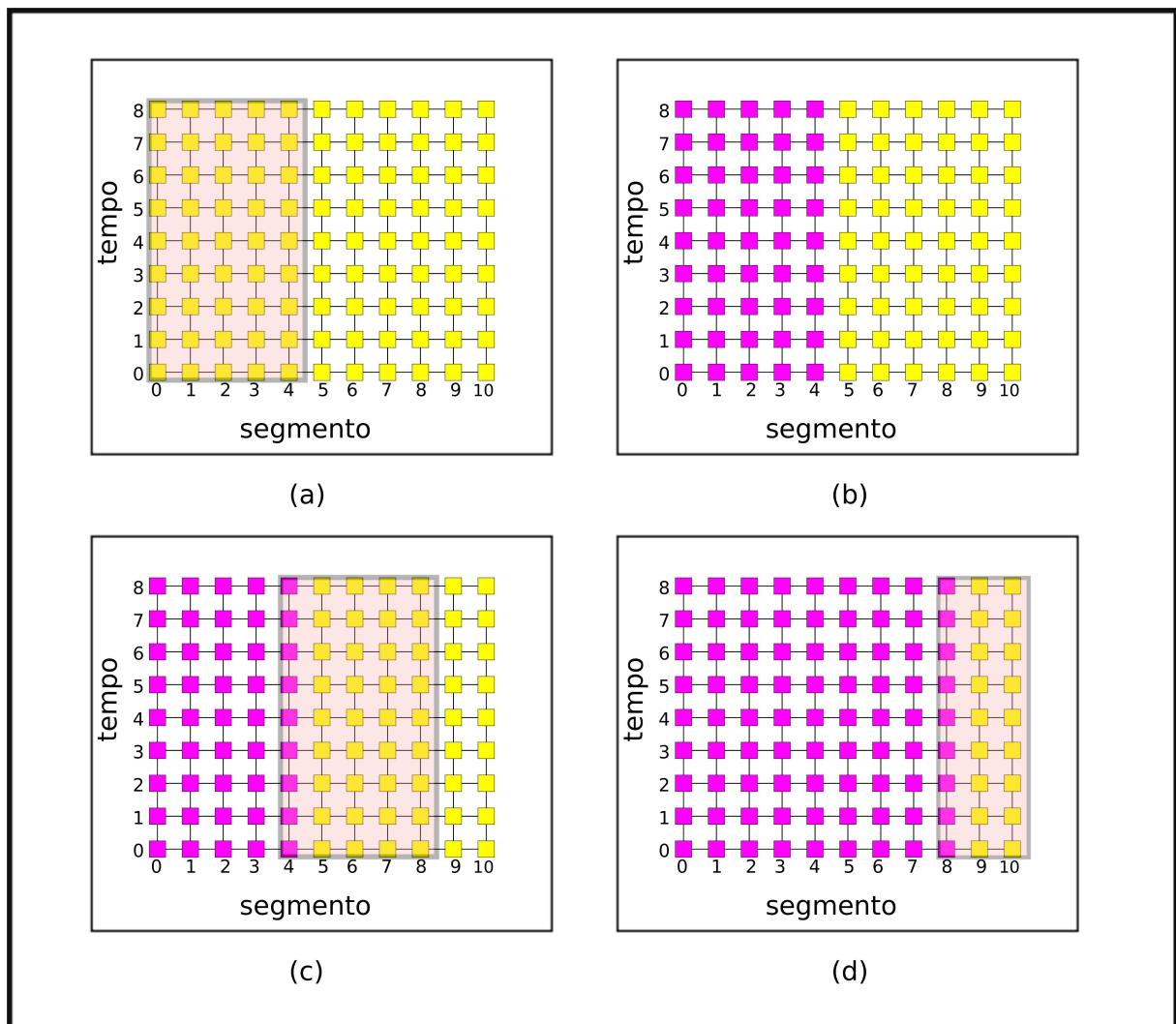


Figura 27: Sequência de cálculos realizados em blocos na implementação CUDA. Após a seleção do bloco, ocorre o cálculo dos segmentos, usando sincronização. Em seguida, um novo bloco é escolhido de forma que contenha a última coluna de segmentos do bloco anterior, para, assim, os segmentos serem calculados.

Na implementação na arquitetura CUDA, foi preciso dar atenção ao uso correto das memórias, como alertado em [23] e [22], levando-se em consideração as restrições e necessidades de cada variável.

Foi utilizado a memória global para as variáveis contidas nas matrizes, isto é, Pressão (H), Vazão (Q) e Tempo (T). Como a memória compartilhada (*shared memory*) é mais rápida que a memória global e, principalmente, permite a sincronia entre as *threads*, ela foi empregada para os vetores intermediários dentro do *kernel*, como é mostrado no trecho de código da Figura 28.

```
__global__ void calc_HQ_RegimeTransiente(_HQT *dMatrizHQT) {
    ...
    extern __shared__ _HQ tempoAtual[ ];
    extern __shared__ _HQ tempoProximo[ ];
    ...
}
```

Figura 28: Trecho de código que exibe aplicação da *shared memory*

Outras variáveis, mostradas na Figura 29, foram alocadas na memória constante.

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

```

__device__ unsigned 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__ unsigned int get_controle() { return controle; }

__device__ unsigned int get_nthreads() { return nthreads; }

};

__constant__ __device__ ConstantesPrograma ConstanteDeviceConstantes;

...

void calc_HQ_GPU() {

    ...

    ConstantesPrograma *host_constantes = (ConstantesPrograma
*)malloc(sizeof(ConstantesPrograma));

    host_constantes->ns = ns;

    host_constantes->hr = hr;

    host_constantes->r = r;

    host_constantes->b = b;

    host_constantes->dt = dt;

    host_constantes->tmax = tmax;

    host_constantes->controle = controle;

    host_constantes->t = t;

    host_constantes->qo = qo;

    host_constantes->nthreads = nthreads;

    ...

    HANDLE_ERROR( cudaMemcpyToSymbol(ConstanteDeviceConstantes,
host_constantes, sizeof(ConstantesPrograma)) );

    free(host_constantes);
}

```

Figura 29: Trecho de código que exibe aplicação da memória constante

Buscou-se alcançar nessa implementação CUDA, além da confiabilidade dos resultados, uma otimização do uso do *device* através do melhor uso das memórias, de modo que fosse evitado ao máximo a troca de dados entre a GPU e o *host*. Dessa maneira, garantiu-se melhor desempenho mesmo nos trechos de programa com maiores restrições, pois a placa de vídeo ficava ocupada na maior parte do tempo.

### **3 SIMULAÇÃO E RESULTADOS**

As simulações foram realizadas em um equipamento com um processador AMD Ryzen 3900X com 12 *cores*, com 32 GB de memória RAM DDR4 e GPU Nvidia RTX2070 com 8GB de memória DDR6.

Foram modelados dois dutos reais, com dados obtidos no site da CETESB, e um duto fictício, sendo que foram processados por três algoritmos. Um algoritmo que utiliza a arquitetura CUDA, outro, convencional, que utiliza apenas um *core* do processador, apelidado de *singlecore*, e finalmente outro algoritmo que se utiliza da arquitetura de processadores de múltiplos núcleos, chamado de *multicore*.

Para apurar a confiabilidade na solução alcançada nos três algoritmos, realizou-se uma simulação de alta granularidade e o resultado obtido nas três versões, quando foram comparados, eram idênticos.

#### **3.1 Cenário com detalhamento de 1 metro**

Deseja-se simular e analisar os problemas propostos em segmentos de aproximadamente 1 metro.

Entretanto, essa abordagem gera malhas computacionais grandes demais para serem alocadas na placa de vídeo. Para contornar o problema, sendo que o foco é o tempo de

processamento entre as plataformas, manipulou-se o problema para serem divididos e executados sequencialmente para replicar a execução do mesmo de forma contínua.

Comparando-se a execução particionada com a contínua da solução *singlecore* e *multicore*, mostraram-se consistentes em tempo de execução comprovando a hipótese.

Assim, foi necessário primeiro dividir cada problema em n partes menores, de modo que pudessem ser simuladas na placa de vídeo e executadas n vezes. Somando-se o tempo de execução sequencial das partes conclui-se o tempo da resolução do problema na sua totalidade.

Primeiro, simulou-se os problemas com os parâmetros conforme a Tabela 3. Note que nessa simulação, a segmentação (n) é calculada considerando-se obter segmentos de aproximadamente 1 metro.

Tabela 3: Parâmetros de entrada da simulação dos problemas com granularidade 1m

Parâmetros	Problema Fictício	Duto OSPLAN II	Duto OPASA 10pol
hr	150	150	150
f	0.035	0.035	0.035
l	120	154	99
d	0.3	0.4572	0.254
n	100	154	100
tmax	240	240	240
a	1030	1030	1030
cdao	0.0009	0.0009	0.0009

O *script* executa a solução para os três arquivos de entrada com os valores dos problemas, já considerando que o problema deve ser dividido em partes menores que serão resolvidos sequencialmente.

Primeiramente é executado uma fração do problema, para se determinar o melhor número de *threads* e outros parâmetros de execução. Dando sequência, a aplicação executa a simulação do transiente hidráulico, e o tempo de execução é aferido e apresentado em tela.

A Tabela 4 a seguir reúne os tempos medidos de cada problema simulado em cada algoritmo:

Tabela 4: Tempos medidos na simulação do cenário om granularidade de 1m

	<b>Problema Fictício</b>	<b>Duto OSPLAN II</b>	<b>Duto OPASA 10pol</b>
<b>CUDA</b>	04m 46,413481s	08m 36,001423s	05m 40,085335s
<b>Singlecore</b>	09m 06,930833s	16m 47,164221s	11m 04,577266s
<b>Multicore</b>	24m 32,887817s	34m 26,423119s	28m 02,528895s

É incontestável que os tempos de execução nas simulações de todos os problemas são expressivamente menores na arquitetura CUDA.

Da fórmula de Speedup (3.1), obtém-se o valor da métrica de comparação entre o desempenho de processamento serial com multicore e CUDA. Abaixo segue o cálculo do Speedup para o problema fictício:

$$S_p = \frac{T_1}{T_p} = \frac{T_{singlecore}}{T_{CUDA}} = \frac{546,930833s}{286,413481s} \rightarrow S_{pCUDA} \cong 1,9096$$

$$S_p = \frac{T_1}{T_p} = \frac{T_{singlecore}}{T_{multicore}} = \frac{546,930833s}{1472,887817s} \rightarrow S_{pmulticore} \cong 0,3713$$

Abaixo segue a Tabela 5 com os valores de Speedup calculados para todos os problemas.

Tabela 5: Cálculos de Speedup para o cenário de granularidade 1m

<b>Speedup</b>	<b>Problema Fictício</b>	<b>Duto OSPLAN II</b>	<b>Duto OPASA 10pol</b>
<b>Singlecore vs CUDA</b>	1,9096	1,9530	1,9541
<b>Singlecore vs Multicore</b>	0,3713	0,4877	0,3950

Para todos os problemas, percebe-se que o Speedup na implementação CUDA é maior que 1. Isto indica que a performance do processamento na arquitetura CUDA é superior à serial (*singlecore*). No problema “OPASA 10pol” ocorre o maior aumento de velocidade, onde há um ganho de 95,41% no desempenho.

O Speedup menor do que zero na Tabela 5 indica que o desempenho da simulação em *multicore* é menor do que em *singlecore*. Isso pode ser explicado por otimizações

implementadas no uso do cache na solução *singlecore*, bem como problemas de sincronização e *race-conditions* na solução *multicore*.

### 3.2 Cenário com detalhamento de 10 centímetros

Essa diferença de tempos fica mais gritante ao diminuir a granularidade do problema para 10cm, como pode ser visto nas Tabela 6 e Tabela 7 abaixo:

Tabela 6: Parâmetros de entrada da simulação dos problemas com granularidade 10cm para *singlecore* e *multicore*

Parâmetros	Problema Fictício	Duto OSPLAN II	Duto OPASA 10pol
hr	150	150	150
f	0.035	0.035	0.035
l	120000	154000	98800
d	0.3	0.4572	0.254
n	1200000	1540000	988000
tmax	240	240	240
a	1030	1030	1030
cdao	0.0009	0.0009	0.0009

Tabela 7: Parâmetros de entrada da simulação dos problemas com granularidade 10cm para CUDA

Parâmetros	Problema Fictício	Duto OSPLAN II	Duto OPASA 10pol
hr	150	150	150
f	0.035	0.035	0.035
l	12	15,3	9,9
d	0.3	0.4572	0.254
n	120	153	99
tmax	240	240	240
a	1030	1030	1030
cdao	0.0009	0.0009	0.0009

Nesse cenário, o problema *multicore* e *singlecore* foram executados de forma integral, aproveitando-se que o sistema operacional garante a alocação da memória dinamicamente, mesmo acima da memória física do computador. Apenas o algoritmo CUDA precisou ser segmentado. Dessa forma, a superioridade do algoritmo *multicore* sobre o *singlecore* fica mais

evidente, mitigando problemas que surgiram no cenário anterior. A Tabela 8 a seguir dispõe os resultados obtidos:

Tabela 8: Tempos medidos na simulação do cenário om granularidade de 10cm

	<b>Problema Fictício</b>	<b>Duto OSPLAN II</b>	<b>Duto OPASA 10pol</b>
<b>CUDA</b>	11h 5m 27,362719s	14h 6m 1,351986s	9h 11m 30,839117s
<b>Singlecore</b>	1d 1h 12m 42,237302s	18h 0m 13,474337s	18h 15m 33,282112s
<b>Multicore</b>	14h 59m 01,668581s	15h 35m 15,115383s	10h 57m 3,137886s

Novamente, da fórmula de Speedup (3.1), obtém-se o valor da métrica de comparação entre o desempenho de processamento serial com *multicore* e CUDA, compilados na Tabela 9.

Tabela 9: Cálculos de Speedup para o cenário de granularidade 10cm

<b>Speedup</b>	<b>Problema Fictício</b>	<b>Duto OSPLAN II</b>	<b>Duto OPASA 10pol</b>
<b>Singlecore vs CUDA</b>	2,2731	1,2768	1,9864
<b>Singlecore vs Multicore</b>	1,6826	1,1550	1,6674

Mais uma vez, o desempenho do CUDA é maior que 1 para todos os cenários, reforçando que sua performance é superior à serial (*singlecore*). No problema fictício ocorre o maior aumento de velocidade, onde há um ganho de 127,31% no desempenho.

Desta vez, o processamento *multicore* foi superior ao *singlecore*, como esperado. Destaque para o Speedup no duto “OPASA 10pol”, que indica um ganho no desempenho do multiprocessamento em 66,74%. Esses tempos menores de execução reforçam que é essencial a adaptação do programa para funcionar de forma otimizada no hardware que realizará o processamento.

## 4 CONCLUSÃO

Os resultados obtidos nesse estudo mostraram que diversos fatores influenciam no desempenho computacional em problemas de transientes hidráulicos.

A arquitetura do processador deve ser levada em consideração e é preciso ter domínio da sua forma de funcionamento. Um *hardware* aperfeiçoado para melhor desempenho em paralelismo não terá uma boa performance se o problema não for modelado de forma a se aproveitar dessa arquitetura, e vice e versa.

No *multicore* percebeu-se que a administração e divisão de tarefas nos *cores* do processador pode tomar um tempo expressivo, fazendo com que a performance de chamadas em paralelo em processadores multinúcleos seja pior que a em chamadas sequenciais no mesmo *hardware*.

Também deve ser avaliado se o problema a ser processado pode ser programado para que funcione de forma otimizada no *hardware* disponível. Problemas maiores ou que utilizem muitos cálculos que dependem do resultado anterior não são recomendados para serem resolvidos de forma paralela, e sim sequencial.

A arquitetura CUDA apresentou resultados expressivamente melhores em todos os cenários simulados. Mesmo otimizando-se o algoritmo para tirar o máximo de proveito da arquitetura de múltiplos núcleos, a organização da arquitetura CUDA é obviamente superior ao que é entregue nos computadores de uso geral. Esses resultados reforçam a importância de o programa ser escrito de maneira a aproveitar ao máximo a arquitetura de *hardware* utilizada.

Apesar de algumas limitações, os resultados mostram que o uso da linguagem CUDAC e de placas gráficas em simulações de transientes hidráulicos são mais eficientes, justificando, inclusive, a adaptação de um programa já existente para que seja processado de forma otimizada nessa arquitetura.

Para sistemas relativamente pequenos, e/ou simulações esporádicas, o custo computacional não se torna um problema. No entanto, quando se deseja 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.

As conclusões obtidas nesse trabalho podem ser consideradas para outros tipos de sistemas a fim de obter ganhos de desempenho computacional.

## 5 APÊNDICE

transitorio.cu

```
#include "cuda.h"
#include "common/book.h"
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <locale.h>

#ifndef LINUX
#include <getopt.h>
#include <sys/time.h>
#include <unistd.h>
#include <wait.h>
#else
#include <ctype.h>
#define __GNU_LIBRARY__
#define MESTRADOCUDA
#include "gettimeofday_win.h"
#include "getopt.h"

#endif

#define PI 3.14159265
#define G 9.806

float hr;
float f;
float l;           // Tamanho horizontal do pipe em metros
float d;           // Diametro do pipe
float dt;          // Delta T
```

```
float tmax;           // Tempo máximo da Simulação em segundos (Parametro
bom pra mexer)

float cdao;

float r;
float a;             // Celeridade da Onda
float qo;
float b;
float cm;
float cp;
float dx;            // Delta X (Comprimento do tubo pelo número de
segimentos)
float tf;

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

struct _HQT {
    float H;
    float Q;
    float T;
};

struct _HQ {
    float H;
    float Q;
};

struct _HQT *matrizHQT;          //Vetor da Strutura H (Pressão)
Q(Vazão) e T(Tempo)
struct _HQT *_dMatrizHQT;        //Matriz da Strutura H (Pressão)
Q(Vazão) e T(Tempo) em device
```

```
struct _HQ *atualHQ;
struct _HQ *proximolHQ;

unsigned int n;           //Número de segmentos
unsigned int N;           //Numeros de segmentos * Delta T
unsigned int ns;          // Numero de segmentos + segmentos montante
+ segmentos justante
unsigned int linha;

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

    __device__ unsigned 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__ unsigned int get_controle() { return controle; }
    __device__ unsigned int get_nthreads() { return nthreads; }
};

__constant__ __device__ ConstantesPrograma ConstantDeviceConstantes;
```

```
/* Globals set by command line args */

int verbosity = 0; /* print trace if set */

int exibe_tempos = 0;

unsigned int nblocks; //Número de blocos que vão ser chamados
unsigned int nthreads; //Número de threads por blocos que vão ser
chamados

unsigned int multiplicador = 1;

int gravarDisco = 0;
char* narq_e = NULL;
char* narq_s = NULL;

char narqe_default[] = "entrada.dat";
char narqs_default[] = "saida.dat";

// NUM Threads
int chunk = 512;

enum _modo {SINGLE = 0, MULTI = 1, CUDA = 2};
enum _modo modo = SINGLE;

extern int paralelo = 0;

//Inicializa vetor com ZEROS
void initZero(float *vector, int size) {
    int i;
    for (i = 0; i < size; i++) {
        vector[i] = 0.0;
    }
}

void initZeroHQT(struct _HQT *vector, int size) {
    int i;
```

```

#pragma omp parallel for if(paralelo)
for (i = 0; i < size; i++) {
    vector[i].H = 0.0;
    vector[i].Q = 0.0;
    vector[i].T = 0.0;
}
}

void initZeroHQ(struct _HQ *vector, int size) {
    int i;
#pragma omp parallel for if(paralelo)
for (i = 0; i < size; i++) {
    vector[i].H = 0.0;
    vector[i].Q = 0.0;
}
}

/*
 * printUsage - Print usage info
 */
void printUsage(char* argv[]) {
    printf("Uso: %s [-hv] -e <file> -s <file> -m 0|1|2 -t <num> -b <num>
-d <num> -p -t\n", argv[0]);
    printf("Options:\n");
    printf(" -h           Print this help message.\n");
    printf(" -v           Optional verbose flag.\n");
    printf(" -e <file>   Arquivo de entrada.\n");
    printf(" -s <file>   Arquivo de saída com o resultado.\n");
    printf(" -m <modo>   0 = Singlecore | 1 = Multicore | 2 = CUDA:
Utiliza a versão indicada.\n");
    printf(" -t <num>     Número de Threads.\n");
    printf(" -d <num>     Número de vezes que vai rodar o simulação
(Simula o tamanho do problema).\n");
    printf(" -p           Roda as instâncias do problema de forma
paralela.\n");
    printf(" -t           Exibe os tempos apenasa.\n");
}

```

```

printf("\nExamples:\n");

    printf(" > %s -v -e entrada.dat -s resuldado.dat -m 2 -b 16 -t 512
-d 5000\n", argv[0]);

    printf(" > %s -v -e entrada.dat -s resuldado.dat -m 1 -c 512 -d
5000\n", argv[0]);

    printf(" > %s -v -e entrada.dat -s resuldado.dat -m 0 -d 5000\n",
argv[0]);

    exit(0);

}

void printInitParam() {

    printf("Processando com os parametros:\n");

    printf("\t Arquivo de entrada: %s\n", narq_e);

    printf("\t Arquivo de resultados: %s\n", narq_s);

    printf("\t Verbosity: %d\n", verbosity);

    printf("\t Modo (0 = Singlecore | 1 = Multicore | 2 = CUDA): %d\n",
modo);

    printf("\t Threads OpenMP: %d\n", chunk);

    printf("\t Threads Cuda: %d\n", nthreads);

    printf("\t Problema aumentado em: %d\n", multiplicador);

}

void printInitMoc() {

    printf("Dados Iniciais:\n");

    printf("\tElementos da matriz = %d\n", N);

    printf("\tlinha = %d\n", linha);

    printf("\ttmax = %4.2f\n", tmax);

    printf("\tdt = %4.2f\n", dt);

    printf("\tdx = %4.2f\n", dx);

    printf("\tl = %f\n", l);

    printf("\tn = %d\n", n);

    printf("\tns = %d\n", ns);

    printf("\tN = %d\n", N);

}

```

```

//Grava o resultado obtido no arquivo definido via parâmetro
void gravaHQTDisco(struct _HQT *matrizHQT, int tamanho) {
    if (gravarDisco == 0) {
        return;
    }

    FILE *arq_s;      // Arquivo de Saída
    if (verbosity) printf("Abrindo %s..\n", narq_s);
    arq_s = fopen(narq_s, "w");

    if (arq_s == NULL) {
        if (verbosity) perror ("Erro ao abrir arquivo de saida");
        #ifndef LINUX
        if (verbosity) printf( "Value of errno: %d\n", errno );
        #endif
    } else {
        if (verbosity) printf("Gravando saída\n");
        int i;
        for(i = 0; i < tamanho; i++) {
            fprintf(arq_s, "%4.2f      %4.5f      %4.5f\n",
matrizHQT[i].T, matrizHQT[i].H, matrizHQT[i].Q);
        }
    }

    fclose(arq_s);
}

gravarDisco = 0; //Só preciso gravar 1 resultado
}

__device__ void printConstantes(ConstantesPrograma *ct) {
    printf("Constantes no DEVICE: \n");
    printf("\tns %d", ct->ns);
    printf("\tqo %4.5f", ct->qo);
    printf("\thr %4.5f", ct->hr);
    printf("\tr %4.5f", ct->r);
    printf("\tb %4.5f", ct->b);
}

```

```

printf("\tt %4.5f", ct->t);
printf("\tdt %4.5f", ct->dt);
printf("\ttmax %4.5f", ct->tmax);
printf("\tcontrole %d", ct->controle);
printf("\tnthreads %d", ct->nthreads);

}

//Cálculo do MOC (single core)
void calc_HQ_CPU();

//Cálculo do MOC (multi core)
void calc_HQ_MCPU();

//Cálculo do MOC (GPU)
__global__ void calc_HQ_RegimeTransiente(_HQT *_dMatrizHQT);
void calc_HQ_GPU();

int main(int argc, char* argv[]) {

    //Dá pau pra mostrar o tempo
    //setlocale(LC_ALL, "pt_BR.UTF-8");

    char str[100];           //Buffer

    float tcpu, tfunc;
    struct timeval p1start, p1stop, totalt;
    struct timeval k1start, k1stop, ktotalt;

    narq_e = narge_default;
    narq_s = narqs_default;
    gravardisco = 0;
    chunk = 512;
    verbosity = 0;
    modo = SINGLE;
}

```

```
multiplicador = 1;
nthreads = 512;
paralelo = 0;
exibe_tempos = 0;

char c;
int temp_mode = 0;

while ( (c = getopt (argc, argv, "e:s:m:c:t:d:pvhk")) != -1) {

    switch(c) {
        case 'e':
            narq_e = optarg;
            break;
        case 's':
            narq_s = optarg;
            gravarDisco = 1;
            break;
        case 'm':
            temp_mode = atoi(optarg);
            if (temp_mode > 2) {
                printUsage(argv);
                exit(1);
            }
            break;
        case 't':
            nthreads = atoi(optarg);
            chunk = nthreads;
            break;
        case 'd':
            multiplicador = atoi(optarg);
            break;
        case 'v':
            verbosity = 1;
            break;
        case 'k':
```

```
        exibe_tempo = 1;
        break;
    case 'p':
        paralelo = 1;
        break;
    case 'h':
        printUsage(argv);
        exit(0);
    default:
        printUsage(argv);
        exit(1);
    }

}

if (temp_mode > 0 && temp_mode < 3) {
    modo = MULTI;
    if (temp_mode > 1)
        modo = CUDA;
}

if (verbosity) printInitParam();

if (verbosity)
    printf("Inicio do processamento\n");

//Leitura dos Dados
FILE *arq_e;      // Arquivo de Entrada
arq_e = fopen(arq_e, "r");
if (arq_e == NULL) {
    if (verbosity) {
        perror ("Erro ao abrir arquivo");
        #ifndef LINUX
        printf( "Value of errno: %d\n", errno );
        #endif
    }
}
```

```

    }

    exit(1);

} else {

    //fscanf(arq_e, "%s", str);

    fscanf(arq_e, "%f", &hr);

    fscanf(arq_e, "%s", str);

    fscanf(arq_e, "%f", &f);

    fscanf(arq_e, "%s", str);

    fscanf(arq_e, "%f", &l);

    fscanf(arq_e, "%s", str);

    fscanf(arq_e, "%f", &d);

    fscanf(arq_e, "%s", str);

    fscanf(arq_e, "%d", &n);

    fscanf(arq_e, "%s", str);

    fscanf(arq_e, "%f", &tmax);

    fscanf(arq_e, "%s", str);

    fscanf(arq_e, "%f", &a);

    fscanf(arq_e, "%s", str);

    fscanf(arq_e, "%f", &cdao);

    fclose(arq_e);

}

if (verbosity) {

    printf("Carregou o arquivo com os dados:\n");

    printf("\tthr = %4.2f\n", hr);

    printf("\t\tf = %4.2f\n", f);

    printf("\t\t(comprimento) l = %4.2f\n", l);

    printf("\t\t(diametro) d = %4.5f\n", d);

    printf("\t\t(Segmentos) n = %d\n", n);

    printf("\t\t(Tempo Max) tmax = %4.2f\n", tmax);

    printf("\t\t(celeridade) a = %4.2f\n", a);

    printf("\t\tcdao = %4.5f\n", cdao);

}

```

```
//Limpa Contadores CPU
timerclear(&p1start);
timerclear(&p1stop);

gettimeofday(&p1start, NULL);

int cont;
timerclear(&k1start);
timerclear(&k1stop);

gettimeofday(&k1start, NULL);
for (cont = 0; cont < multiplicador; cont++) {
    //printf("Rodando %d modo %d\n", cont, modo);
    switch(modo) {
        case SINGLE:
            calc_HQ_CPU();
            break;
        case MULTI:
            calc_HQ_MCPU();
            break;
        case CUDA:
            //printf("Cuda %d\n", cont);
            calc_HQ_GPU();
            break;
    }
}

getttimeofday(&k1stop, NULL);
timersub(&k1stop, &k1start, &ktotalt);
tfunc = 0.001 * (ktotalt.tv_sec * 1000000 + ktotalt.tv_usec);

if (exibe_tempos)
    printf("Tempo transitorio = %f ms\n", tfunc);
```

```

gettimeofday(&p1stop, NULL);

//Gasto de tempo ms
timersub(&p1stop, &p1start, &totalt);

tcpo = 0.001 * (totalt.tv_sec * 1000000 + totalt.tv_usec);

if (exibe_tempos)
    printf("==> Tempo total de execução = %f ms, Modo %d ==>\n",
tcpo, modo);

}

void calc_HQ_CPU() {
    int i;

    int controle;      //Controle para linkar vetores H e Q as matrizes
    //Controla o pulo para as linhas

    // Constantes //
    as = (PI * d * d) / 4;
    dx = 1 / n;
    dt = dx / a;
    ns = n + 2; //segmentos + montante + justrante
    r = f * dx / (2 * G * d * as * as);
    b = a/(G * as);

    //Alocação Vetor e Matrizes
    linha = (int)(tmax / dt);
    N = ns + (ns * linha); //Delta T para 0 + conjunto de delta T +
    ultima iteração Delta T

    if (verbosity) printInitMoc();

    controle = 0;
}

```

```

size_t sizeMATRIX = N; /* sizeof(float);

size_t sizeVECTOR = ns; /* sizeof(float);

matrizHQT = (struct _HQT *)malloc(sizeof(struct _HQT) * sizeMATRIX);

atualHQ = (struct _HQ *)malloc(sizeof(struct _HQ) * sizeVECTOR);
proximolHQ = (struct _HQ *)malloc(sizeof(struct _HQ) * sizeVECTOR);

// Inicializa H e Q

initZeroHQT(matrizHQT , sizeMATRIX);

initZeroHQ(atualHQ, sizeVECTOR);
initZeroHQ(proximolHQ, sizeVECTOR);

t = 0;

//Cálculo do Regime Permanente inicial
if (verbosity) printf("Regime Permanente inicial\n");
qo = sqrt(((cdao * cdao) * 2 * hr * hr)/(1 + (cdao * cdao) * 2 * G *
n * r));

for(i = 0; i < ns; i++) {
    atualHQ[i].H = hr - (i) * r *qo * qo;
    atualHQ[i].Q = qo;
    matrizHQT[i].H = atualHQ[i].H;
    matrizHQT[i].Q = atualHQ[i].Q;
    matrizHQT[i].T = t;
}

//Inicio Transitório
while (t < tmax) {
    t = t + dt;
    controle++;
}

```

```

//Pontos interiores;

for(i = 1; i < ns; i++) {

    cp = atualHQ[i-1].H + b * atualHQ[i-1].Q;
    bp = b + r * fabs(atualHQ[i-1].Q);
    cm = atualHQ[i+1].H - b * atualHQ[i+1].Q;
    bm = b + r * fabs(atualHQ[i+1].Q);
    proximolHQ[i].Q = (cp - cm) / (bp + bm);
    proximolHQ[i].H = cp - bp * proximolHQ[i].Q;
}

//Condição de contorno de montante: Reservatório

proximolHQ[0].H = hr;
cm = atualHQ[1].H - b * atualHQ[1].Q;
bm = b + r * fabs(atualHQ[1].Q);
proximolHQ[0].Q = (hr - cm)/bm;

//condição de contorno de jusante: Válvula

cp = atualHQ[ns-2].H + b * atualHQ[ns-2].Q;
proximolHQ[ns-1].Q = 0;
proximolHQ[ns-1].H = cp;

//Atualização

for (i = 0; i < ns; i++) {
    atualHQ[i].H = proximolHQ[i].H;
    atualHQ[i].Q = proximolHQ[i].Q;

    unsigned int ind = (controle * ns) + i;
    if (ind < sizeMATRIX) {
        matrizHQT[ind].H = atualHQ[i].H;
        matrizHQT[ind].Q = atualHQ[i].Q;
        matrizHQT[ind].T = t;
    }
}

```

```

    }

}

if (verbosity) printf("Processamento Completo\n");
//Impressão completa
gravaHQTDisco(matrizHQT, sizeMATRIX);

free(matrizHQT);

free(atualHQ);
free(proximolHQ);
}

void calc_HQ_MCPU() {
    int i;
    int linha;
    int controle;      //Controle para linkar vetores H e Q as matrizes
    (Controla o pulo para as linhas)

    // Constantes //
    as = (PI * d * d) / 4;
    dx = l / n;
    dt = dx / a;
    ns = n + 2; //segmentos + montante + justrante
    r = f * dx / ( 2 * G * d * as * as);
    b = a/(G * as);

    //Alocação Vetor e Matrizes
    //linha = ((int)(tmax/dt))+1;;
    //N = ns + (ns * linha)+ns; //Delta T para 0 + conjunto de delta T +
    ultima iteração Delta T
    linha = (int)(tmax / dt);
    N = ns + (ns * linha);
}

```

```

if (verbosity) printInitMoc();

controle = 0;

size_t sizeMATRIX = N;
size_t sizeVECTOR = ns;

matrizHQT = (struct _HQT *)malloc(sizeof(struct _HQT) * sizeMATRIX);

atualHQ = (struct _HQ *)malloc(sizeof(struct _HQ) * sizeVECTOR);
proximolHQ = (struct _HQ *)malloc(sizeof(struct _HQ) * sizeVECTOR);

// Inicializa H e Q

initZeroHQT(matrizHQT , sizeMATRIX);

initZeroHQ(atualHQ, sizeVECTOR);
initZeroHQ(proximolHQ, sizeVECTOR);

t = 0;

//Cálculo do Regime Permanente inicial
if (verbosity) printf("Regime Permanente inicial\n");
qo = sqrt(((cdao * cdao) * 2 * hr * hr)/(1 + (cdao * cdao) * 2 * G *
n * r));

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

```

```

        matrizHQT[i].T = t;
    }

}

//Inicio Transitório

while (t < tmax) {
    if (verbosity) printf("Passo T %f de %f\n", t, tmax);
    t = t + dt;
    controle++;

#pragma omp parallel shared(atualHQ,proximolHQ,r,b,ns,chunk)
private(i,cm,bm,cp,bp)
{
    #pragma omp for schedule(dynamic,chunk) nowait
    //Pontos interiores;
    for(i = 1; i < ns; i++) {
        cp = atualHQ[i-1].H + b * atualHQ[i-1].Q;
        bp = b + r * fabs(atualHQ[i-1].Q);
        cm = atualHQ[i+1].H - b * atualHQ[i+1].Q;
        bm = b + r * fabs(atualHQ[i+1].Q);
        proximolHQ[i].Q = (cp - cm) / (bp + bm);
        proximolHQ[i].H = cp - bp * proximolHQ[i].Q;
    }
}
//Condição de contorno de montante: Reservatório

proximolHQ[0].H = hr;
cm = atualHQ[1].H - b * atualHQ[1].Q;
bm = b + r * fabs(atualHQ[1].Q);
proximolHQ[0].Q = (hr - cm)/bm;

//condição de contorno de jusante: Válvula

cp = atualHQ[ns-2].H + b * atualHQ[ns-2].Q;
proximolHQ[ns-1].Q = 0;

```

```

proximolHQ[ns-1].H = cp;

//Atualização

#pragma omp parallel
shared(atualHQ,proximolHQ,matrizHQT,ns,t,chunk) private(i)

{
    #pragma omp for schedule(dynamic,chunk) nowait
    for (i = 0; i < ns; i++) {
        atualHQ[i].H = proximolHQ[i].H;
        atualHQ[i].Q = proximolHQ[i].Q;
        unsigned int ind = (controle * ns) + i;
        if (ind < sizeMATRIX) {
            matrizHQT[ind].H = atualHQ[i].H;
            matrizHQT[ind].Q = atualHQ[i].Q;
            matrizHQT[ind].T = t;
        }
    }

}

if (verbosity) printf("Processamento Completo\n");
//Impressão completa
gravaHQTDisco(matrizHQT, sizeMATRIX);

free(matrizHQT);

free(atualHQ);
free(proximolHQ);
}

void calc_HQ_GPU() {
    //int linha;
    int controle;      //Controle para linkar vetores H e Q as matrizes
    (Controla o pulo para as linhas)
}

```

```

// Constantes //
as = (PI * d * d) / 4;
dx = l / n;
dt = dx / a;
ns = n + 2; //segmentos + montante + justrante
r = f * dx / ( 2 * G * d * as * as);
b = a/(G * as);

//Alocação Vetor e Matrizes
linha = (int)(tmax / dt);
N = ns + (ns * linha); //Delta T para 0 + conjunhto de delta T +
ultima iteração Delta T

//printf("Uso da memória %d para N=%d \n", (int)(sizeof(struct _HQT)
* N), N);
if (verbosity) printInitMoc();

controle = 0;

size_t sizeMATRIX = N; /* sizeof(float);

matrizHQT = (_HQT *)malloc(sizeof(_HQT) * sizeMATRIX);

// Inicializa H e Q
initZeroHQT(matrizHQT , sizeMATRIX);
//Aloca vetores na memória do device
HANDLE_ERROR( cudaMallocManaged(&_dMatrizHQT, (sizeof(_HQT) *
sizeMATRIX)) );

//Dados : Host -> Device
HANDLE_ERROR( cudaMemcpy(_dMatrizHQT, matrizHQT, (sizeof(_HQT) *
sizeMATRIX), cudaMemcpyHostToDevice) );

//Cálculo do Regime Permanente inicial
t = 0;

```

```

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

    nblocks = (ns + nthreads) / nthreads;

    //no c++ daria pra passar um objeto ?

    ConstantesPrograma *host_constantes = (ConstantesPrograma
*)malloc(sizeof(ConstantesPrograma));

    host_constantes->ns = ns;
    host_constantes->hr = hr;
    host_constantes->r = r;
    host_constantes->b = b;
    host_constantes->dt = dt;
    host_constantes->tmax = tmax;
    host_constantes->controle = controle;
    host_constantes->t = t;
    host_constantes->qo = qo;
    host_constantes->nthreads = nthreads;

    //HANDLE_ERROR( cudaMallocManaged(&ConstantDeviceConstantes,
sizeof(ConstantesPrograma)) );
    HANDLE_ERROR( cudaMemcpyToSymbol(ConstantDeviceConstantes,
host_constantes, sizeof(ConstantesPrograma)) );
    free(host_constantes);

    cudaEvent_t start, stop;

    HANDLE_ERROR( cudaEventCreate(&start) );
    HANDLE_ERROR( cudaEventCreate(&stop) );
    HANDLE_ERROR( cudaEventRecord(start, 0) );

    calc_HQ_RegimeTransiente<<<nblocks, nthreads,
nthreads*sizeof(_HQ)>>>(_dMatrizHQT);

    HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
    HANDLE_ERROR( cudaEventSynchronize( stop ) );

```

```

    float elapsedTime;

    HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime, start, stop ) );

    if (exibe_tempos)
        printf("Tempo do Kernel = %f ms\n", elapsedTime);

    //Dados : Device -> Host
    HANDLE_ERROR( cudaMemcpy(matrizHQT, _dMatrizHQT, (sizeof(_HQT) * sizeMATRIX), cudaMemcpyDeviceToHost) );

    if (verbosity) printf("Processamento Completo\n");
    //Impressão completa
    gravaHQTDisco(matrizHQT, sizeMATRIX);

    free(matrizHQT);
    cudaFree(_dMatrizHQT);
}

__global__ void calc_HQ_RegimeTransiente(_HQT *_dMatrizHQT) {
/*
int x = threadIdx.x + blockIdx.x * blockDim.x;
int y = threadIdx.y + blockIdx.y * blockDim.y;
int offset = x + y * blockDim.x * gridDim.x;
*/

float _cp = 0.0;
float _bp = 0.0;
float _cm = 0.0;
float _bm = 0.0;

unsigned int _ns = ConstantDeviceConstantes.get_ns();
float _qo = ConstantDeviceConstantes.get_qo();
float _hr = ConstantDeviceConstantes.get_hr();
float _r = ConstantDeviceConstantes.get_r();
float _b = ConstantDeviceConstantes.get_b();

```

```
float _t = ConstantDeviceConstantes.get_t();
float _dt = ConstantDeviceConstantes.get_dt();
float _tmax = ConstantDeviceConstantes.get_tmax();
unsigned int _nthreads = ConstantDeviceConstantes.get_nthreads();

unsigned int _controle = 0;

//printConstantes(&ConstantDeviceConstantes);

int _i = threadIdx.x + blockIdx.x * blockDim.x;
int indice = (int)(_i % _nthreads);

extern __shared__ _HQ tempoAtual[];
extern __shared__ _HQ tempoProximo[];

//Regime Permanente inicial
if (_i < _ns) {
    tempoAtual[indice].H = _hr - (_i) * _r *_qo * _qo;
    tempoAtual[indice].Q = _qo;

    __syncthreads();

    _dMatrizHQT[(_controle * _ns) + _i].H = tempoAtual[indice].H;
    _dMatrizHQT[(_controle * _ns) + _i].Q = tempoAtual[indice].Q;
    _dMatrizHQT[(_controle * _ns) + _i].T = _t;
}

//Regime Transitório
if (_i < _ns) {

    while (_t < _tmax) {
        _t = _t + _dt;
        _controle++;

        //Pontos interiores;
```

```

        if (_i < _ns && _i >= 1) {
            _cp = tempoAtual[indice-1].H + _b * tempoAtual[indice-1].Q;
            _bp = _b + _r * fabs(tempoAtual[indice-1].Q);
            _cm = tempoAtual[indice+1].H - _b * tempoAtual[indice+1].Q;
            _bm = _b + _r * fabs(tempoAtual[indice+1].Q);
            tempoProximo[indice].Q = (_cp - _cm) / (_bp + _bm);
            tempoProximo[indice].H = _cp - _bp * tempoProximo[indice].Q;
        }

        //Condição de contorno de montante: Reservatório
        if (_i == 0) {
            tempoProximo[0].H = _hr;
            _cm = tempoAtual[1].H - _b * tempoAtual[1].Q;
            _bm = _b + _r * fabs(tempoAtual[1].Q);
            tempoProximo[0].Q = (_hr - _cm)/_bm;
        }

        //condição de contorno de jusante: Válvula
        if (_i == _ns - 1) {
            _cp = tempoAtual[_ns-2].H + _b * tempoAtual[_ns-2].Q;
            tempoProximo[_ns-1].Q = 0;
            tempoProximo[_ns-1].H = _cp;
        }

        //Atualização
        tempoAtual[indice].H = tempoProximo[indice].H;
        tempoAtual[indice].Q = tempoProximo[indice].Q;

        __syncthreads();

        _dMatrizHQT[(_controle * _ns) + _i].H = tempoAtual[_i].H;
    }
}

```

```

        _dMatrizHQT[(_controle * _ns) + _i].Q = tempoAtual[_i].Q;
        _dMatrizHQT[(_controle * _ns) + _i].T = _t;

    }

}

}

```

## simula.py

```

import psutil
import platform
from datetime import datetime
from subprocess import call, check_output, run, STDOUT, CalledProcessError
import datetime
import os
import os.path

def get_size(bytes, suffix="B"):
    """
    Scale bytes to its proper format
    e.g:
        1253656 => '1.20MB'
        1253656678 => '1.17GB'
    """
    factor = 1024
    for unit in ["", "K", "M", "G", "T", "P"]:
        if bytes < factor:
            return f"{bytes:.2f}{unit}{suffix}"
        bytes /= factor

def executaComando(parametro):
    start = datetime.datetime.now()
    try:

```

```

exec_com = ["./transitorio"] + parametro
if os.name == 'nt':
    exec_com = ["transitorio.exe"] + parametro
# print(exec_com)
#tmp = check_output(exec_com)
str_exec = ' '.join(map(str, exec_com))
return_code = call(str_exec, shell=True)
if (return_code != 0):
    print("Error: ", return_code)
    return datetime.datetime.max - start
elapsed_time = datetime.datetime.now() - start
return elapsed_time

except CalledProcessError as ex:
    #print ("Error: ", ex, ex.output, ex.returncode)
    return datetime.datetime.max - start


def testaSimulacao(param, threads_list):
    menor_tempo = 0.0
    menor_vector = datetime.datetime.now(), threads_list[0]
    for threads in threads_list:
        #print("Threads: ", threads)
        parametro = param + ["-t", threads]
        elapsed_time = executaComando(parametro)
        if (menor_tempo == 0.0):
            menor_tempo = elapsed_time
        #print ("Resultado GPU Cuda")
        #print (tmp)
        #print("Segundos: ",
elapsed_time.seconds,":",elapsed_time.microseconds)
        if (elapsed_time < menor_tempo):
            menor_tempo = elapsed_time
            menor_vector = elapsed_time, threads
    return menor_vector

```

```

def grava_best_param(arq_entrada, threads_mult, threads_cuda):

    file1 = open(arq_entrada + ".simul", "w")
    file1.writelines(threads_mult + "\n")
    file1.writelines(threads_cuda + "\n")
    file1.close()

def obtem_best_param(arq_entrada):

    file1 = open(arq_entrada + ".simul", "r+")
    threads_mult = file1.readline()
    threads_cuda = file1.readline()
    file1.close()

    return threads_mult, threads_cuda

def simulacao(mult_problem_ini, mult_problem_final, arq_entrada,
param=False):

    parametro_ini = ["-e", arq_entrada, "-d", mult_problem_ini]
    if (param):
        parametro_ini = ["-e", arq_entrada, "-d", mult_problem_ini, "-p"]

    if os.path.exists(arq_entrada + ".simul") == False:
        print("Testando Single Core")
        parametro = parametro_ini + ["-m", "0"]
        elapsed_time = executaComando(parametro)
        print("Single Segundos: {}".format(elapsed_time))

        print("Testando CUDA")
        threads_list = list()
        for i in range(1, 14):

```

```

    threads_list.append(str(pow(2, i)))

parametro = parametro_ini + ["-m", "2"]
menor_vector_cuda = testaSimulacao(parametro, threads_list)
tempinho_cuda, threads_cuda = menor_vector_cuda
print("Menor thread: " + threads_cuda +
      " tempo {}".format(tempinho_cuda))

print("Testando Multi")
threads_list = list()
for i in range(1, 14):
    threads_list.append(str(pow(2, i)))

parametro = parametro_ini + ["-m", "1"]
menor_vector_mult = testaSimulacao(parametro, threads_list)
tempinho_mult, threads_mult = menor_vector_mult
print("Menor thread: " + threads_mult +
      " tempo {}".format(tempinho_mult))

grava_best_param(arq_entrada, threads_mult, threads_cuda)

threads_mult, threads_cuda = obtem_best_param(arq_entrada)

print("*"*20, "Processamento Final:", "*"*20)
parametro_ini = ["-e", arq_entrada, "-d", mult_problem_final, "-v"]

best = "NONE"
menor_tempo = datetime.timedelta.max

### MULTI ###
parametro = parametro_ini + ["-m", "1", "-t", threads_mult]
elapsed_time = executaComando(parametro)

if (elapsed_time < menor_tempo):
    menor_tempo = elapsed_time

```

```

best = "MULTI"

print("Tempo Multi: {}".format(elapsed_time))

os.system("pause")

### SINGLE ###

parametro = parametro_ini + ["-m", "0"]
elapsed_time = executaComando(parametro)

if (elapsed_time < menor_tempo):
    menor_tempo = elapsed_time
    best = "SINGLE"

print("Tempo Single: {}".format(elapsed_time))

return best, menor_tempo

### CUDA ###

parametro = parametro_ini + ["-m", "2", "-t ", threads_cuda]
# print(parametro)
elapsed_time = executaComando(parametro)

if (elapsed_time < menor_tempo):
    menor_tempo = elapsed_time
    best = "CUDA"

print("Tempo CUDA: {}".format(elapsed_time))

return best, menor_tempo

print("*"*40, "System Information", "*"*40)
uname = platform.uname()
print(f"System: {uname.system}")
print(f"Node Name: {uname.node}")
print(f"Release: {uname.release}")
print(f"Version: {uname.version}")
print(f"Machine: {uname.machine}")

```

```

print(f"Processor: {uname.processor}")

# let's print CPU information
print("*"*40, "CPU Info", "*"*40)
# number of cores
print("Physical cores:", psutil.cpu_count(logical=False))
print("Total cores:", psutil.cpu_count(logical=True))

#####
# Estratégia: Problema Mestrado Grande (chamadas sequenciais)
mult_problem_inicial = "1"
mult_problem_final = "1"
arq_entrada = "entrada_gg.dat"

best, menor_tempo = simulacao(
    mult_problem_inicial, mult_problem_final, arq_entrada, False)
print("Best: " + best + " tempo {}".format(menor_tempo))

# Estratégia: Problema Mestrado Grande (chamadas paralelas)
#best, menor_tempo = simulacao(mult_problem_inicial, mult_problem_final,
#arq_entrada, True)
#print ("Best: " + best + " tempo {}".format(menor_tempo))

#####
# Estratégia: Problema Duto OSPLAN II (chamadas sequenciais)
mult_problem_inicial = "1"
mult_problem_final = "1"
arq_entrada = "entrada_OSPLAN_II.dat"

best, menor_tempo = simulacao(
    mult_problem_inicial, mult_problem_final, arq_entrada, False)

```

```
print("Best: " + best + " tempo {}".format(menor_tempo))

#print("*10, "Estratégia: Problema Duto OSPLAN II (chamadas
#paralelas)", "*10)

#best, menor_tempo = simulacao(mult_problem_inicial, mult_problem_final,
#arq_entrada, True)

#print ("Best: " + best + " tempo {}".format(menor_tempo))

print("*10, "Estratégia: Problema Duto OPASA 10pol (chamadas
sequenciais)", "*10)

mult_problem_inicial = "1"
mult_problem_final = "1"
arq_entrada = "entrada_OPASA_10.dat"

best, menor_tempo = simulacao(
    mult_problem_inicial, mult_problem_final, arq_entrada, False)
print("Best: " + best + " tempo {}".format(menor_tempo))

exit(0)
```

## 6 REFERÊNCIAS BIBLIOGRÁFICAS

- [1] ELECTRIC POWER RESEARCH INSTITUTE (EPRI), “Water Hammer Handbook for Nuclear Plant Engineers and Operators”, 1996
- [2] IBIBLIO CATALOG, GPGPU: General Purpose Computation on Graphics Hardware”, disponível em <https://www.ibiblio.org/catalog/items/show/3990>. Acesso em Agosto de 2020.
- [3] 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”, 2011.
- [4] CROUS, P. A., VAN ZYL, J. E. e NEL, A., “Using stream processing to improve the speed of hydraulic network solvers”, 2008.
- [5] ALMEIDA, A.B., “Protecção Contra o Golpe de Aríete, Manual de Saneamento Básico”, Vol. 1, 1990.
- [6] STREETER, V.L., WYLIE, E.B., “Fluid Mechanics”, edição 8, McGraw-Hill, 1985.
- [7] KOELLE, E., “Transientes hidráulicos em condutos forçados, aplicações em engenharia”, Escola Politécnica da USP, 1983.
- [8] ROCHA, M.S., “Influência do fator de atrito no cálculo do transiente hidráulico”, Tese de Mestrado, Universidade Estadual de Campinas (UNICAMP), 1998.
- [9] CONTRUÁGIL, “Golpe de Aríete”, disponível em <<https://www.construagil.com.br/post/golpe-de-ariete>>. Acesso em janeiro de 2020.
- [10] IZQUIERDO J. e IGLESIAS P. L., “Mathematical Modelling of Hydraulic Transients in Simple Systems”, 2002.
- [11] WOOD D. J. , LINGIREDDY, S. , BOULOS P. , KARNEY B.W. e MCPHERSON D. L., “Numerical methods for modeling transient flow in distribution systems”, 2005.
- [12] ABNT, “NBR12215 – Projeto de adutora de água para abastecimento público - Procedimento”, volume I, 1992.
- [13] 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”, 2008.
- [14] KHRONOS GROUP, “Open standard for parallel programming of heterogeneous systems”, disponível em: <<http://www.khronos.org/opencl/>>. Acesso em novembro de 2019.

- [15] OPENMP, “OpenMP API specification for parallel programming”, disponível em: <<https://www.openmp.org/spec-html/5.0/openmp.html>>. Acesso novembro de 2019.
- [16] NASCIMENTO JÚNIOR, O. S., “Técnicas de computação paralela aplicadas ao Método das Características em Sistemas Hidráulicos”, Dissertação de mestrado, Unicamp, 2013
- [17] ALMASI G. S. e GOTTLIEB A., “Highly parallel computing”. Benjamin-Cummings Publishing Co., EUA, 1989.
- [18] TANEMBAUM, A. S., “Organização Estruturada de Computadores”, Pearson/Prentice Hall, 2006.
- [19] FLYNN M. J., “Some computer organizations and their effectiveness”, 1972.
- [20] GROPP, W. e LUSK, E., “The Message Passing Interface (MPI) standard”, disponível em: <<http://www.mcs.anl.gov/research/projects/mpi/>>. Acesso janeiro de 2020.
- [21] VERDIÈRE G. C., “Introduction to GPGPU, a hardware and software background”. Académie des sciences, 2010.
- [22] SANDERS J., KANDROT E., “CUDA by example: an introduction to general-purpose GPU programming”, 2011.
- [23] NVIDIA CORPORATION, “NVIDIA CUDA Programming C Guide”, 2011.
- [24] CETESB, “Dutos no Estado de São Paulo”, disponível em: <<https://cetesb.sp.gov.br/emergencias-quimicas/tipos-de-acidentes/dutos/dutos-no-estado-de-sao-paulo/>>. Acesso em janeiro de 2020.
- [25] TANEMBAUM, A. S., “Sistemas Operacionais Modernos”. Pearson, São Paulo, SP, 2007.
- [26] IKEDA P. A., “Um estudo do uso eficiente de programas em placas gráficas”. Dissertação de Mestrado, 2011.
- [27] KAN S.H., “Metrics and Models in Software Quality Engineering”, Adison Wesley, 2002.
- [28] SOUZA, P.A., “Escoamento Transitório em Conduto Forçado. Golpe de Aríete”, Escola Politécnica da USP.