

Pedro Henrique dos Santos Cunha

Calculo da equação de onda em duas dimensões utilizando CUDA

Projeto final referente à disciplina de Computação Científica de Alto Desempenho II que visa resolver a equação de onda em duas dimensões utilizando técnicas de paralelismo em GPGPUs.

Universidade Federal Fluminense - UFF

Campus Volta Redonda

Instituto de Ciências Exatas

Brasil

26 de julho de 2022

Resumo

Alguns problemas de natureza física e matemática precisam ser resolvidos computacionalmente. No entanto, dependendo do tipo de problema, a resolução pode demorar muito ou gastar uma quantidade excessiva de memória. Com o intuito de melhorar o desempenho das resoluções de problemas que utilizam muitos dados, foram desenvolvidas uma série de soluções em computação utilizando técnicas de paralelismo. Abordaremos, neste trabalho, uma solução com base em técnicas de paralelismo voltadas à GPGPUs utilizando a API CUDA para o cálculo da solução do problema abordado.

O intuito de escolhermos esse tema se deve ao fato do problema ter uma gama de possíveis implementações computacional, uma vez que a API utilizada tem se expandido no mercado de trabalho e também no meio acadêmico, além de se tratar de um problema físico de interesse do aluno.

Palavras-chaves: CUDA. Paralelização. Equação de onda.

Abstract

Some problems of a physical and mathematical nature need to be solved computationally. However, depending on the type of problem, resolution may take a long time or consume an excessive amount of memory. In order to improve the performance of problem solving that uses a lot of data, a series of solutions were developed in computing using parallelism techniques. We will approach, in this work, a solution based on parallelism techniques at GPGPUs using the CUDA API to calculate the solution of the problem choosed.

The purpose of choosing this theme is due to the fact that the problem has a range of possible computational implementations, since the API used has expanded in the job market and also in the academic environment, in addition to being a physical problem of interest to the student.

Key-words: CUDA. Parallelization. Wave equation.

Sumário

Sumário	4
Lista de ilustrações	6
1 INTRODUÇÃO	8
2 IMPLEMENTAÇÃO E METODOLOGIA UTILIZADA	9
3 BENCHMARK - OTIMIZAÇÃO A NÍVEL DE COMPILAÇÃO	12
3.1 Profile do programa	13
3.2 Comparação dos resultados obtidos utilizando as flags de compilação para o caso serial	13
4 IMPLEMENTAÇÃO DO PROGRAMA UTILIZANDO CUDA	16
4.1 Utilização da API	16
4.2 Escalabilidade	17
4.3 Cálculo da equação de onda em CUDA	19
4.3.1 Profiling da execução	19
4.3.2 Benchmark [GeForce GTX 1650] - Flags de otimização em CUDA	20
4.3.2.1 Serial Case x No Flag CUDA	20
4.3.2.2 Flags Case	22
4.3.3 Benchmark [NVIDIA GeForce RTX 2060]	26
4.3.4 Benchmark [NVIDIA Tesla K40]	28
4.3.5 Benchmark [NVIDIA Tesla V100]	29
5 VALIDAÇÃO DOS RESULTADOS	32
6 CONCLUSÕES	33
REFERÊNCIAS	34
7 APÊNDICES	35
7.1 Apêndice A - Visualização da solução para diferentes malhas e condições iniciais	35
7.1.1 A.1 - Malha 5×5 e $N = 100$	35
7.1.2 A.2 - Malha 10×10 e $N = 100$	36
7.1.3 A.3 - Malha 50×50 e $N = 1000$	36

8	ANEXOS	37
8.1	A - Código serial	37
8.2	B - Implementação da API CUDA	42

Listas de ilustrações

Figura 1 – Ilustração do método das diferenças finitas. Clique aqui para acessar a fonte da figura.	9
Figura 2 – Fluxograma do programa	11
Figura 3 – Condição inicial utilizada no problema.	11
Figura 4 – Profile do programa SERIAL	13
Figura 5 – Comparação do tempo de execução do programa serial para cada conjunto de flags	14
Figura 6 – Comparação da eficiência do programa serial para cada conjunto de flags	14
Figura 7 – Comparação do speedup do programa serial para cada conjunto de flags	15
Figura 8 – Esquematização do fluxo de processamento de dados em CUDA.	17
Figura 9 – Arquitetura de GPUs baseada na arquitetura Kepler.	18
Figura 10 – Arquitetura de GPUs baseadas na arquitetura Ada.	18
Figura 11 – Profile da implementação em CUDA	20
Figura 12 – Comparação do tempo de execução para o caso serial e a implementação CUDA sem as flags de otimização.	21
Figura 13 – Eficiência atingida em termos de tempo de execução da implementação do CUDA em contraste com o caso serial, sem flags.	21
Figura 14 – Visualização dos dados obtidos para o tempo de execução em cada caso utilizando as flags de otimizações.	22
Figura 15 – Visualização da eficiência atingida em termos do tempo de execução em cada caso utilizando as flags de otimizações.	23
Figura 16 – Visualização do speedup obtido em termos do tempo de execução em cada caso utilizando as flags de otimizações.	24
Figura 17 – Visualização dos melhores dados obtidos para o tempo de execução por malha utilizada.	25
Figura 18 – Comparação dos tempos de execução por malha para a melhor flag utilizada em relação ao programa serial.	25
Figura 19 – Eficiência atingida por malha para a melhor flag utilizada em relação ao programa serial.	26
Figura 20 – Comparação dos tempos de execução por malha utilizando a GPU NVIDIA GeForce RTX 2060 para a melhor flag utilizada em relação ao programa serial.	27
Figura 21 – Eficiência atingida por malha utilizando a GPU NVIDIA GeForce RTX 2060 para a melhor flag utilizada em relação ao programa serial.	27

Figura 22 – Comparaçāo dos tempos de execuāo por malha utilizando a GPU NVIDIA Tesla K40 para a melhor flag utilizada em relaāo ao programa serial.	28
Figura 23 – Eficiêncā atingida por malha utilizando a GPU NVIDIA Tesla K40 para a melhor flag utilizada em relaāo ao programa serial.	29
Figura 24 – Comparaçāo dos tempos de execuāo por malha utilizando a GPU NVIDIA Tesla V100 para a melhor flag utilizada em relaāo ao programa serial.	30
Figura 25 – Eficiêncā atingida por malha utilizando a GPU NVIDIA Tesla V100 para a melhor flag utilizada em relaāo ao programa serial.	30
Figura 26 – Onda calculada no instante final estabelecido. Resultado do cálculo da equaāo de onda utilizando a GPU NVIDIA Geforce GTX 1650.	32

1 Introdução

A equação da onda é uma equação diferencial parcial linear de segunda ordem importante que descreve a propagação das ondas tais como ondas sonoras, luminosas, etc. Surge em áreas como a acústica, eletromagnetismo, e dinâmica dos fluidos e outras importantes áreas da física.

Na sua forma mais simples, a equação de onda diz respeito a uma variável de tempo t e uma ou mais variáveis espaciais associadas à uma função escalar.

O problema a ser abordado se trata da resolução numérica equação de onda em duas dimensões espaciais, dado por:

$$\frac{\partial^2 f}{\partial t^2} = \nu_0^2 \left(\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \right) \quad (1.1)$$

Sujeita a condições de contorno e também condições iniciais. O termo ν_0 é dado por:

$$\nu_0 = \sqrt{\frac{\mu_0}{T_0}}$$

Para resolvemos a equação de onda, utilizaremos o método das diferenças finitas. Dessa forma, se faz necessário a discretização da equação para que possamos tratá-la de forma computacional.

2 Implementação e metodologia utilizada

Em tempo, uma vez que utilizaremos o método das diferenças finitas, precisaremos definir uma região no espaço para o cálculo desse potencial. Se definirmos uma malha marcada por ponto (i, j) para encontrar o valor de um determinado ponto (i, j) usaremos os quatro pontos a sua volta, pois iremos utilizar os sucessores e os antecessores em cada uma das direções.

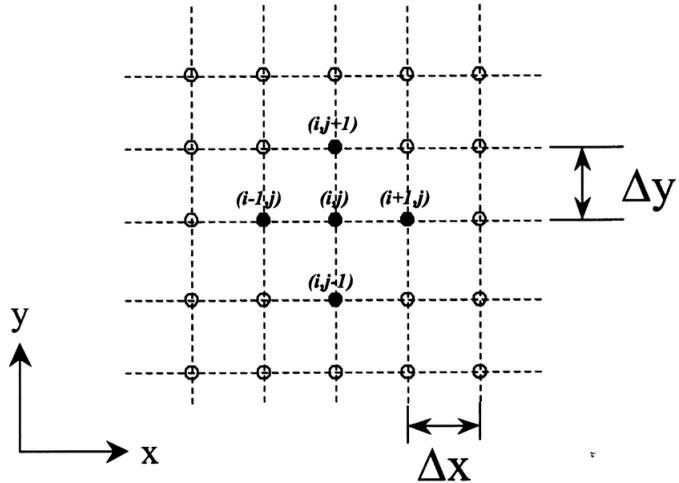


Figura 1 – Ilustração do método das diferenças finitas. [Clique aqui para acessar a fonte da figura.](#)

Aliado a isso, também indexaremos como τ o instante de tempo correspondente ao ponto (i, j) na função de onda, uma vez que ela apresenta dependência temporal. Podemos discretizar as derivadas de ordem 2 a partir da expansão em séries de Taylor nas regiões vizinhas do ponto (i, j) , onde obtemos:

$$\frac{\partial^2 f}{\partial t^2} = \frac{f_{i,j}^{\tau+1} - 2f_{i,j}^{\tau} + f_{i,j}^{\tau-1}}{(\Delta t)^2} \quad (2.1)$$

$$\frac{\partial^2 f}{\partial x^2} = \frac{f_{i+1,j}^{\tau} - 2f_{i,j}^{\tau} + f_{i-1,j}^{\tau}}{(\Delta x)^2} \quad (2.2)$$

$$\frac{\partial^2 f}{\partial y^2} = \frac{f_{i,j+1}^{\tau} - 2f_{i,j}^{\tau} + f_{i,j-1}^{\tau}}{(\Delta y)^2} \quad (2.3)$$

Onde Δx , Δy e Δt são os espaçamentos tomadas na expansão em séries de Taylor, podendo ser interpretados como o espaçamento entre cada ponto da malha (no caso das coordenadas espaciais) e entre cada instante de tempo (no caso da coordenada temporal).

Substituindo as expressões em (1.1), encontramos:

$$f_{i,j}^{\tau+1} = 2f_{i,j}^{\tau} - f_{i,j}^{\tau-1} + \nu_0^2 \frac{(\Delta t)^2}{(\Delta x)^2} (f_{i+1,j}^{\tau} - 2f_{i,j}^{\tau} + f_{i-1,j}^{\tau}) + \nu_0^2 \frac{(\Delta t)^2}{(\Delta y)^2} (f_{i,j+1}^{\tau} - 2f_{i,j}^{\tau} + f_{i,j-1}^{\tau})$$

Por simplicidade, podemos tomar $\Delta x = \Delta y = \Delta t$, no entanto, tal abordagem não modificará a onda e, após imposta a condição de contorno e a condição inicial, a onda oscilará da mesma forma em todos os instantes de tempo, isto é, se a onda inicial for uma senóide, ela não modificará sua natureza, oscilando da mesma maneira no espaço e no tempo, até o final do programa. Dessa forma, definimos:

$$\alpha^2 = \nu_0^2 \frac{(\Delta t)^2}{(\Delta x)^2}$$

$$\gamma^2 = \nu_0^2 \frac{(\Delta t)^2}{(\Delta y)^2}$$

Chegando à formula de recursão discretizada utilizada no calculo da função de onda:

$$f_{i,j}^{\tau+1} = 2f_{i,j}^{\tau} (1 - \alpha^2 - \gamma^2) - f_{i,j}^{\tau-1} + \alpha^2 f_{i+1,j}^{\tau} + \alpha^2 f_{i-1,j}^{\tau} + \gamma^2 f_{i,j+1}^{\tau} + \gamma^2 f_{i,j-1}^{\tau} \quad (2.4)$$

A equação (2.4) define a equação a ser calculada numéricamente pelo nosso programa. A condição de contorno e as condições iniciais serão as condições de Dirichlet homogêneas, isto é, a função de onda será identicamente nula nos limites da malha.

Na figura 2 apresentamos o fluxograma do algoritmo que será utilizado na resolução do problema.

O primeiro problema acontece na etapa da alocação dinâmica de memória devido ao fato de estarmos lidando com um 3D-array, onde para cada elemento de tempo, possuímos um 2d-array (matriz) associado a ele. Ao tomarmos uma malha grande e um grande intervalo de tempo, podemos ter o problema de out of memory, impedindo a execução do programa.

De forma a contornar o problema citado, repare que não é preciso, necessariamente, alojar um grande intervalo de tempo para persistirmos o calculo da função de onda usando diferenças finitas pois precisamos apenas de um ponto no futuro ($\tau + 1$), um ponto no presente (τ) e um ponto no passado ($\tau - 1$). Uma das maneiras de utilizarmos somente 3 pontos no tempo para o cálculo da equação de onda é reciclá-los utilizando o resto da divisão por 3 (%3) da coordenada temporal. Uma outra maneira é utilizarmos, no total, 3 matrizes (3 2d-arrays) e atualizarmos a matriz seguinte com base na anterior para persistirmos os dados temporalmente.

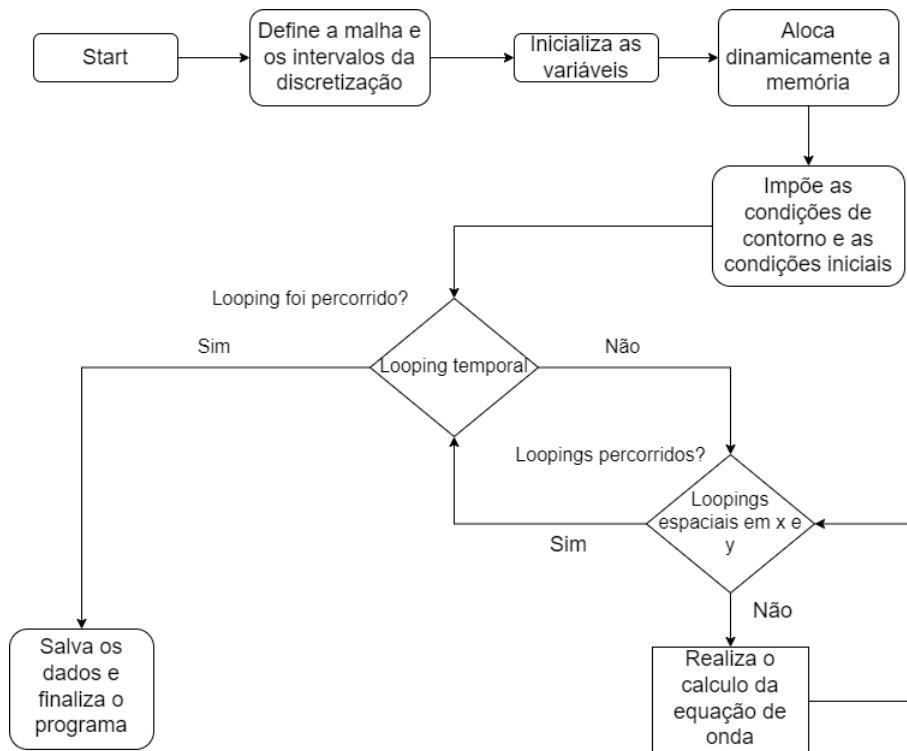


Figura 2 – Fluxograma do programa

A primeira estratégia foi utilizada no cálculo do programa em serial, enquanto que a segunda alternativa foi utilizada no cálculo do programa paralelo usando CUDA.

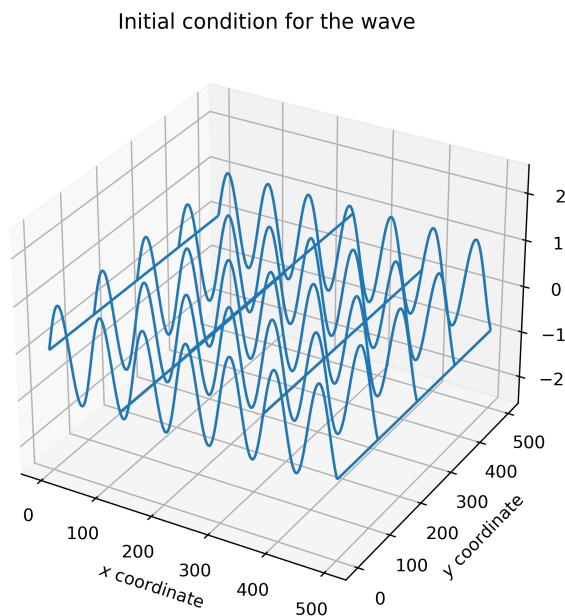


Figura 3 – Condição inicial utilizada no problema.

3 Benchmark - Otimização a nível de compilação

Uma das primeiras técnicas de otimização que podem ser utilizadas, são as técnicas baseadas na mudança das flags de compilação do programa.

Um compilador é um programa que lê um conjunto de instruções de um código fonte e traduz para a linguagem de máquina. Para que isso ocorra, o código passa por diversas fases, como pelo analisador semântico e pelo analisador sintático. Podemos citar como exemplo os compiladores GNU Compiler e iFort da Intel.

Na compilação de um código, existem vários níveis nos quais o compilador pode intervir para otimizar o código $-O0$, $-O1$, $-O2$, $-O3$. Cada um desse níveis habilita uma série de flags de otimização. As flags habilitadas por cada nível dependem do compilador e do próprio computador.

Usaremos o compilador GNU Compiler e iFort da Intel, onde analisaremos o programa sendo executado para uma malha 5000×5000 habilitando as seguintes flags nos 5 casos:

1. SERIAL case, Flags: $-OX$
2. Flags: $-OX$ $-fexpensive-optimizations$ $-m64$ $-foptimize-register-move$ $-funroll-loops$ $-ffast-math$
 $-mtune=native$ $-march=native$
3. Flags: $-OX$ $-fexpensive-optimizations$ $-m64$ $-foptimize-register-move$ $-funroll-loops$ $-ffast-math$
 $-mtune=corei7-avx$ $-march=corei7-avx$
4. Flags: $-OX$ $-w$ $-mp1$ $-zero$ $-xHOST$
5. Flags: $-OX$ $-w$ $-mp1$ $-zero$ $-xHOST$ $-fast=2$

Onde o termo $-OX$ se refere à variação $-O0$, $-O1$, $-O2$, $-O3$.

Utilizamos para esse benchmark, a maquina disponibilizada no laboratório 107C, cujas especificações estão listadas abaixo:

- Intel(R) Core(TM) CPU i7-2600, 3.40GHz. Máx: 3.80 GHz, Min: 1.60 GHz
- 4 núcleos, cada núcleo possuindo 2 Threads.
- 12GB RAM

3.1 Profile do programa

O profile pode ser visualizado utilizando o gprof, onde podemos ver que o maior tempo de execução do programa se refere ao cálculo do método das diferenças finitas.

Flat profile:						
Each sample counts as 0.01 seconds.						
time	%	cumulative	self	self	total	name
99.81	11119.36	11119.36	1	11.12	11.12	finiteDifference
0.00	11119.74	0.37	1	0.00	0.00	writeFiles
0.00	11120.01	0.27	1	0.00	0.00	derivativeCondition
0.00	11120.16	0.15	1	0.00	0.00	allocArray
0.00	11120.30	0.14	1	0.00	0.00	initialCondition
0.00	11120.30	0.00	1	0.00	11.12	actionWork
0.00	11120.30	0.00	1	0.00	0.00	contourCondition

Figura 4 – Profile do programa SERIAL

3.2 Comparação dos resultados obtidos utilizando as flags de compilação para o caso serial

Os resultados encontrados para o tempo podem ser visualizados no gráfico abaixo:

Podemos reparar que nesse caso, o melhor conjunto de flags foi o conjunto de flags 3. Repare que o compilador intel se mostrou eficiente, especialmente com o conjunto de flags 5, demonstrando uma boa estabilidade quando variamos as flags -OX. No entanto, o melhor caso quanto ao tempo de execução ocorreu utilizando a flag -O3 junto do conjunto de flags 3.

Podemos, ainda, encontrar a eficiencia em porcentagem para cada caso utilizando a expressão:

$$\text{Eficiência} = \left(1 - \frac{T[n]}{T_0}\right) \times 100 \quad (3.1)$$

Onde $T[n]$ é o tempo para cada flag -OX. Os resultados encontrados estão dispostos no gráfico abaixo:

Novamente, com base nos dados obtidos, o melhor caso fora obtido utilizando os conjuntos de flags 2 e 3, onde obtivemos a maior eficiência utilizando o conjunto 3 com -O3, sendo próxima de 72%.

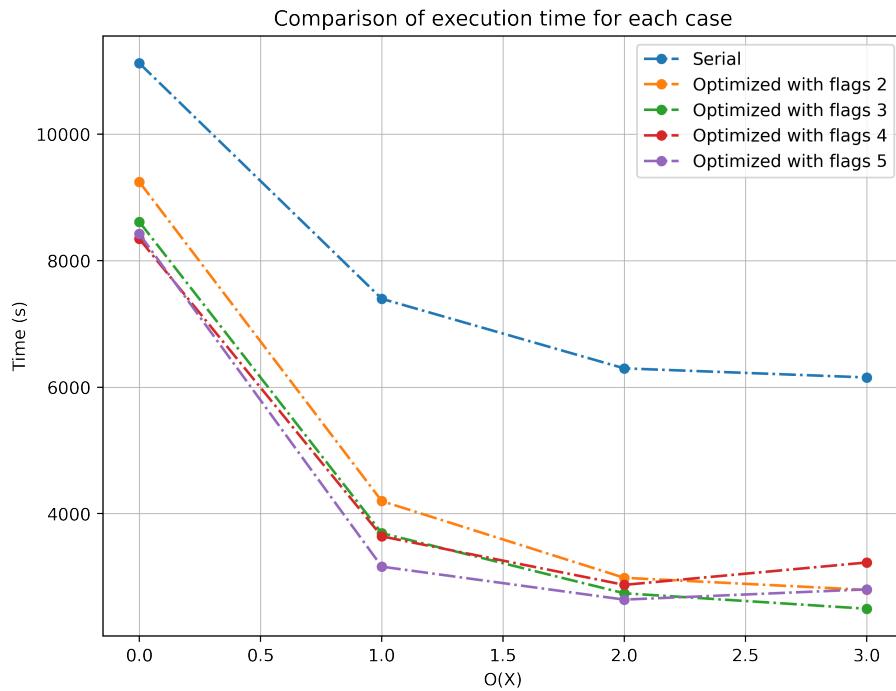


Figura 5 – Comparação do tempo de execução do programa serial para cada conjunto de flags

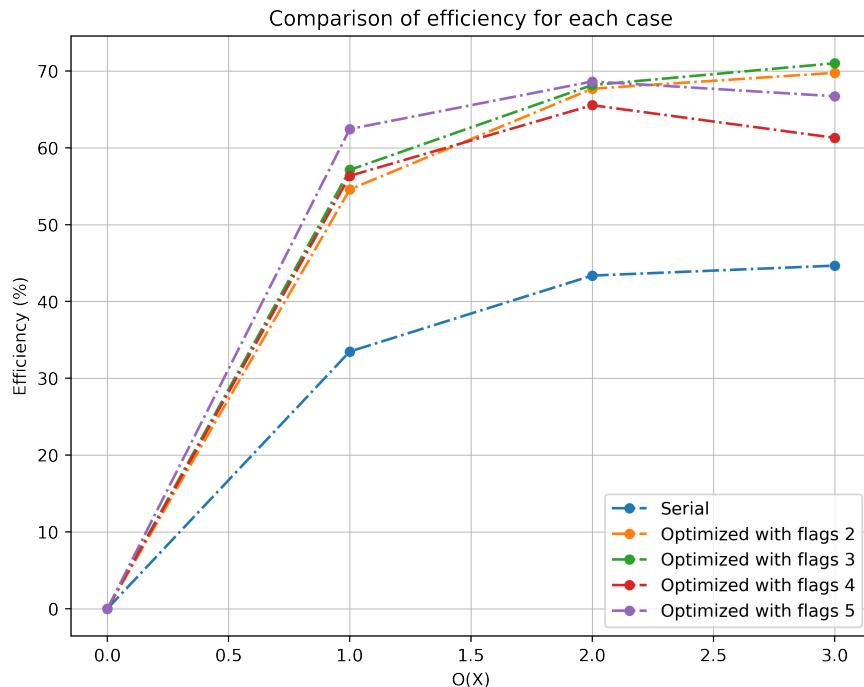


Figura 6 – Comparação da eficiência do programa serial para cada conjunto de flags

Mas podemos, ainda, analisar o speedup para cada caso. Nesse sentido, analisamos os dados obtidos utilizando a seguinte expressão:

$$\text{Speedup} = \frac{T_0}{T[n]} \quad (3.2)$$

Onde novamente $T[n]$ é o tempo para cada flag -OX. Os resultados obtidos foram:

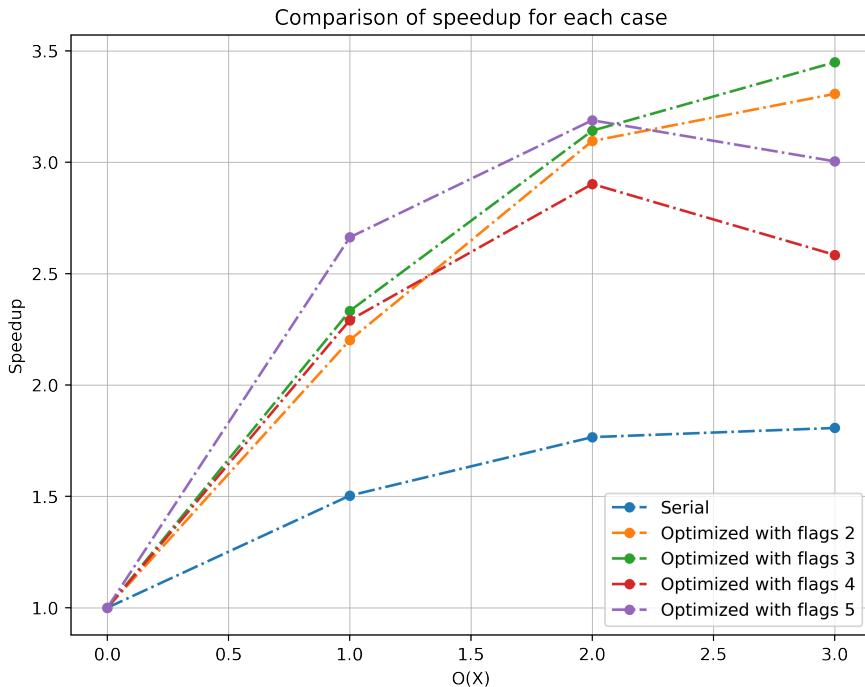


Figura 7 – Comparação do speedup do programa serial para cada conjunto de flags

Repare que embora o conjunto de flags 3 tenha obtido o maior speedup quando utilizado a flag -O3, o conjunto de flags que possui um melhor comportamento levando em conta todas as flags -OX é o conjunto de flags 2. A partir disso, podemos aferir a escalabilidade do programa. Repare que a curva do speedup para o conjunto de flags 2 e 3 são bastante parecidas, no entanto, para o conjunto de flags 2, a curva é mais próxima do que seria um crescimento linear.

Ambas as curvas demonstram um bom comportamento para seus respectivos conjuntos de flags, entretanto, o conjunto de flags 3 leva em consideração as flags particulares do processador disponibilizado pelo laboratório 107 -mtune=corei7-avx e -march=corei7-avx.

4 Implementação do programa utilizando CUDA

CUDA, sigla para *Compute Unified Device Architecture*, é uma extensão para a linguagem de programação C e C++, a qual possibilita o uso de computação paralela, isto é, uma API destinada a computação paralela através de GPGPUs, criada pela Nvidia.[\[1\]](#) destinada a placas gráficas que suportem a API (normalmente placas gráficas com chipset da Nvidia). A ideia por trás disso é que programadores possam usar os poderes da unidade de processamento gráfico (GPU) para realizar operações mais rapidamente. A API CUDA dá acesso ao conjunto de instruções virtuais da GPU e a elementos de computação paralela, para a execução do programa em núcleos de computação das GPGPUs.

A GPU se torna uma opção viável para o trabalho de processamento paralelo por ter sido desenvolvida para atender à demanda de processos de computação 3D em alta resolução e em tempo real. Assim, com o passar do tempo, as GPUs modernas se tornaram muito eficientes ao manipular grandes quantidades de informações.

Uma solução utilizando CUDA segue um fluxograma bem padronizado para todas as implementações: Inicialmente, os dados são copiados da memória principal para a GPU. Depois disso, o processador aloca o processo para a GPU, que então executa as tarefas simultaneamente em seus núcleos. Depois disso, o resultado faz o caminho inverso, ou seja, ele é copiado da memória da GPU para a memória principal.

4.1 Utilização da API

A arquitetura das GPUs são divididas em SM's (Streaming Multiprocessors) composto por cores que chamamos de CUDA Cores. O número de CUDA Cores e, consequentemente, do número de SM's de cada GPU é responsável pelo maior e melhor desempenho da mesma para executar as tarefas designadas.

O processamento dos dados na GPGPU ocorre em grids e blocos de threads. Uma thread é uma unidade de processamento, capaz de realizar uma operação simples. Um grid é uma coleção de threads que quando acionadas, executam a função designada pela API em blocos de threads que chamamos de blocks. Os blocks, por sua vez, são um grupo de threads que executam a função mencionada anteriormente em um mesmo SM, onde o SM é uma região da GPGPU que possui uma memoria exclusiva que é compartilhada para todas as threads que fazem parte do block acionado.

Toda função (ou rotina) que será utilizada na API do CUDA é chamada de função

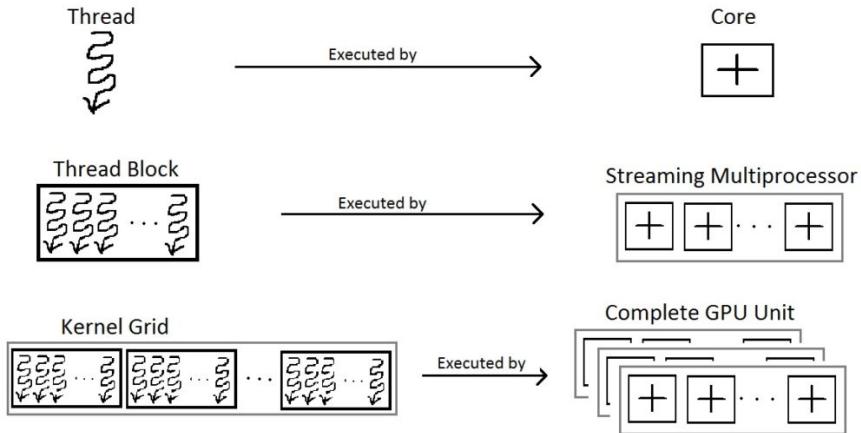


Figura 8 – Esquematização do fluxo de processamento de dados em CUDA.

Kernel. Essa função é definida usando o especificador `__global__` onde passamos como argumento as informações do número de blocos e threads que serão utilizados na função através da sintaxe « `numBlocks, numThreads` ».

Além disso, algumas palavras como Host e Device, no contexto da API do CUDA, se refere à CPU e a GPU, respectivamente.

4.2 Escalabilidade

A escalabilidade de uma implementação em CUDA depende, além de outros fatores, do modelo da GPU. GPUs mais potentes são aquelas que possuem um numero maior de SM's (consequentemente, de CUDA Cores) e também possuem métricas elevadas para cálculo de operações em single precision e double precision. Nesse projeto, utilizaremos as GPUs com as especificações listadas na tabela a seguir:

Modelo GPU	Arquitetura	Número de CUDA Cores	Memória
NVIDIA GeForce GTX 1650	Turing	896	4GB
NVIDIA GeForce RTX 2060	Turing	1920	12GB
NVIDIA Tesla K40	Volta	2880	12GB
NVIDIA Tesla V100	Volta	5120	16GB - 32GB

As GPUs da arquitetura Volta são GPUs para uso profissional em HPC, IA e outras aplicações, enquanto que as GPUs de arquitetura Turing são comumente placas desenvolvidas para melhor performance em jogos, embora ainda possam ser usadas para fins profissionais.

A evolução das famílias e arquiteturas das GPUs da NVIDIA se deram principalmente à capacidade de obter maiores números de SM's com o avanço dos anos. Na figura

[9](#) podemos ver uma esquematização das GPUs baseadas na arquitetura Kepler e na figura [10](#) temos uma esquematização das novas GPUs da SÉRIE RTX 40 baseada na arquitetura Ada



Figura 9 – Arquitetura de GPUs baseada na arquitetura Kepler.



Figura 10 – Arquitetura de GPUs baseadas na arquitetura Ada.

Dessa maneira, esperamos obter uma maior escalabilidade do programa ao utilizarmos os modelos profissionais V100 e K40 disponibilizados no cluster SDumont do LNCC, em contraste com as GPUs de uso pessoal.

4.3 Cálculo da equação de onda em CUDA

4.3.1 Profiling da execução

De modo a analisar os maiores gargalos da implementação, analisamos o profile da execução do programa para diferentes configurações de rotinas kernel.

Fora testado 3 configurações do Kernel:

1. 3D-like: Nessa configuração, fizemos uma esquematização do nosso 3d-array da equação de onda para um 3d-like Kernel no formato $3 \times N \times N$, onde o numero 3 seria a coordenada cíclica do tempo conforme discutido no capítulo 2.

Na ocasião, não foi possível identificar os ganhos de perfomance pelo fato da implementação não ter sido bem sucedida.

2. 2D-like e decomposição espacial: Nessa configuração, a esquematização foi baseada em uma decomposição espacial no domínio da implementação, onde repartimos a malha $N \times N$ em 2^L sub-malhas onde cada sub-malha realizava o cálculo da equação de onda e sincronizava com seus vizinhos.

Os resultados para essa esquematização foram superiores às outras 2 configurações, no entanto, houve uma não-conformidade em relação à solução analítica de até 60% devido a erros de sincronização entre as partições da malha.

3. 1D-like: Como ultima configuração, implementamos o cálculo da equação de onda utilizando a técnica de mapeamento de matrizes em vetores utilizando column-major order. Nesse sentido, utilizamos 3 matrizes (representando os pontos no presente, futuro e passado) onde cada matriz foi configurada como um vetor percorrido pelas colunas.

Sendo essa a melhor configuração devido à sua sincronização intrínseca do CUDA, podemos alcançar uma maior perfomance da execução do programa sem perder de vista a confiabilidade dos resultados do cálculo.

Para traçarmos um profiling, utilizamos uma malha pequena para um intervalo de tempo igualmente pequeno, de forma a analisar os gargalhos da implementação. O compilador do CUDA *nvcc* possui uma ferramenta de profiling próprio que pode ser chamado através do comando *nvprof*, onde devemos dar permissão de administrador para que o mesmo possa analisar as propriedades do device. Os resultados obtidos podem ser visualizados abaixo:

Percebemos, através da figura 11, que o maior gargalo se encontra não no Kernel e no cálculo das diferenças finitas, como mostrado no profile 4, mas sim na cópia das

==22111== Profiling application: ./wave.x						
Type	Time(%)	Time	Calls	Avg	Min	Max
GPU activities:	49.97%	2.51083s	3000	836.94us	760.68us	2.3931ms
	45.69%	2.29606s	3000	765.35us	657.66us	1.6430ms
	4.34%	218.21ms	1000	218.21us	214.52us	254.33us
API calls:	97.60%	5.58821s	6000	931.37us	701.09us	2.9554ms
	2.13%	122.21ms	3	40.738ms	46.663us	122.11ms
	0.26%	15.051ms	1000	15.051us	9.4590us	45.078us
	0.00%	183.73us	3	61.244us	39.887us	103.33us
	0.00%	119.50us	97	1.2320us	100ns	51.693us
	0.00%	95.790us	1	95.790us	95.790us	95.790us
	0.00%	26.194us	1	26.194us	26.194us	26.194us
	0.00%	5.5770us	1	5.5770us	5.5770us	5.5770us
	0.00%	1.6850us	3	561ns	95ns	1.4080us
	0.00%	668ns	1	668ns	668ns	668ns
	0.00%	566ns	2	283ns	98ns	468ns
	0.00%	348ns	1	348ns	348ns	348ns
	0.00%	186ns	1	186ns	186ns	186ns

Figura 11 – Profile da implementação em CUDA

informações entre o Host e o Device, representamos mais de 97% do tempo de execução do programa.

Tal comportamento era esperado. Embora o poder computacional desbloqueado pela implementação em CUDA seja muito alto, a transferência de informações entre o Host e o Device possui uma limitação causada pela rede utilizada no hardware da máquina, tornando tal item do hardware tão vital quanto a própria GPU.[\[2\]](#)

4.3.2 Benchmark [GeForce GTX 1650] - Flags de otimização em CUDA

Da mesma maneira que fora feito no capítulo 3, podemos utilizar flags de otimização para melhor a perfomance da nossa implementação através do compilador *nvcc*.

As flags utilizadas, nesse caso, podem ser direcionadas tanto para a CPU quanto para a GPU utilizando a diretiva *-Xptxas*. As flags utilizadas no benchmark da implementação foram:

- No Flags.
- Flags: *-OX*
- Flags: *-Xptxas -OX*
- Flags: *-use_fast_math -OX*

O benchmark foi feito utilizando a GPU GeForce GTX 1650, que possui no total 896 CUDA Cores. Dessa maneira, para que fosse feito uma melhor análise do real poder de otimização dos cálculos através das flags de otimização, fora fixados tamanhos de malha multiplas de 896, isto é, 896×896 , 1792×1792 , 2688×2688 e 3584×3584 .

Começamos analisando a execução para os casos sem as flags de otimização.

4.3.2.1 Serial Case x No Flag CUDA

Nesse caso, os resultados obtidos para cada uma das malhas está exposto na figura

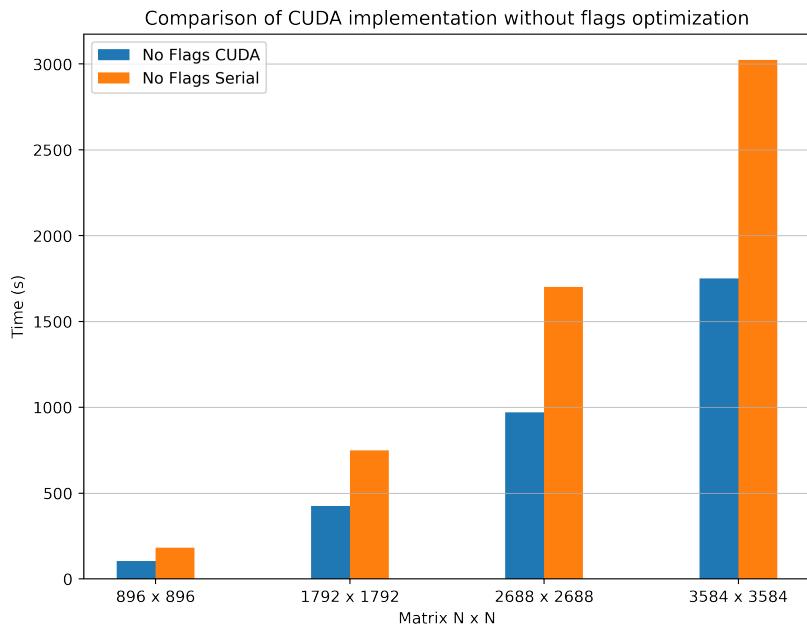


Figura 12 – Comparaçāo do tempo de execuāo para o caso serial e a implementaāo CUDA sem as flags de otimizaāo.

O ganho de perfomance foi substancial em cada malha utilizada. Podemos, ainda, analisar os ganhos de eficiêncāa utilizando a equaāao (3.1) para a implementaāo sem flags do CUDA, de forma a obter uma melhor mētrica para a analise dos dados:

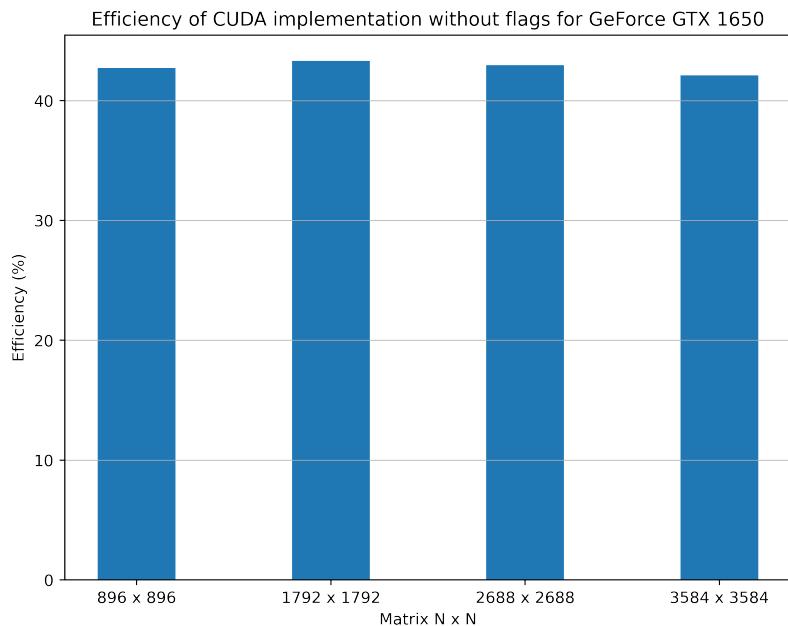


Figura 13 – Eficiêncāa atingida em termos de tempo de execuāo da implementaāo do CUDA em contraste com o caso serial, sem flags.

Em mēdia, alcançamos, tomando a mēdia arimética, 48.27% de eficiêncāa na

implementação do CUDA. No caso onde a malha correspondia ao número exatos de CUDA Cores da GPU utilizada, obtemos um ganho de 64.7% de eficiência em termos de tempo de execução do programa.

4.3.2.2 Flags Case

À fim de melhorar a perfomance de nossa implementação, podemos melhorar nosso tempo de execução utilizando as flags citadas no inicio do capítulo. Tendo como base o tempo da implementação do CUDA sem as flags, alcançamos os seguintes resultados:

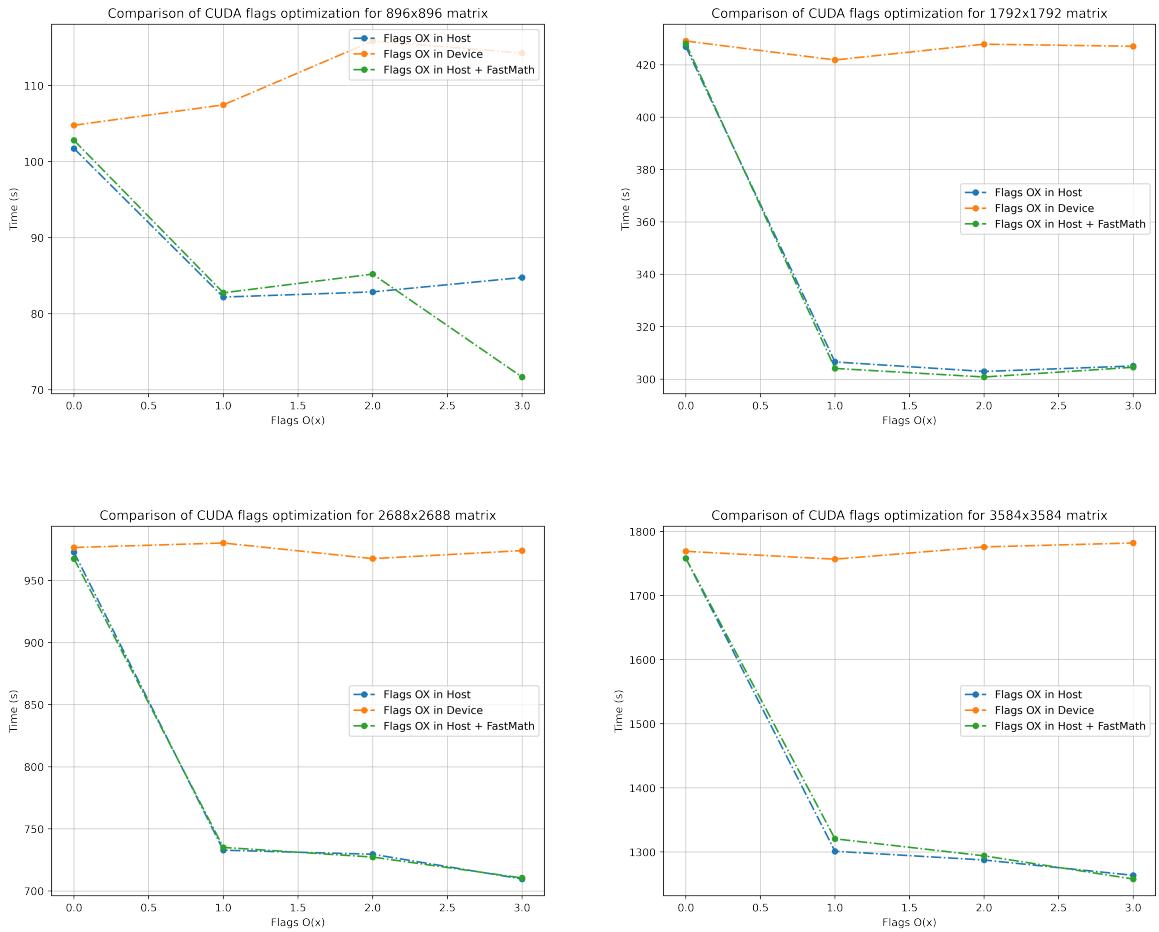


Figura 14 – Visualização dos dados obtidos para o tempo de execução em cada caso utilizando as flags de otimizações.

Notamos que para o caso onde o tamanho da malha não corresponde ao numero de CUDA Cores da GPU, isto é, a otimização via flags de compilação causa efeitos significativos, o que não acontece para o caso em que utilizamos a malha 896×896 .

Uma interpretação para esse fato ocorre devido à natureza da otimização e a configuração do Kernel. Quando usamos a malha 896×896 , o cálculo das diferenças finitas, isto é, do Kernel da aplicação, ocorre quase que instantâneamente. O restante do tempo

da implementação, de acordo com o profiling realizado, se dá devido ao grande numero de sincronizações entre o Host e o Device. As flags de otimizações atuam na execução do código, aumentando a agressividade como o compilador interpreta cada instrução mesmo que possa causar alguns erros se comparado ao programa serial, mas não altera o tempo de execução das funções do CUDA, em principal, a função Memcpy.

Podemos extender nossa análise através dos gráficos de eficiência:

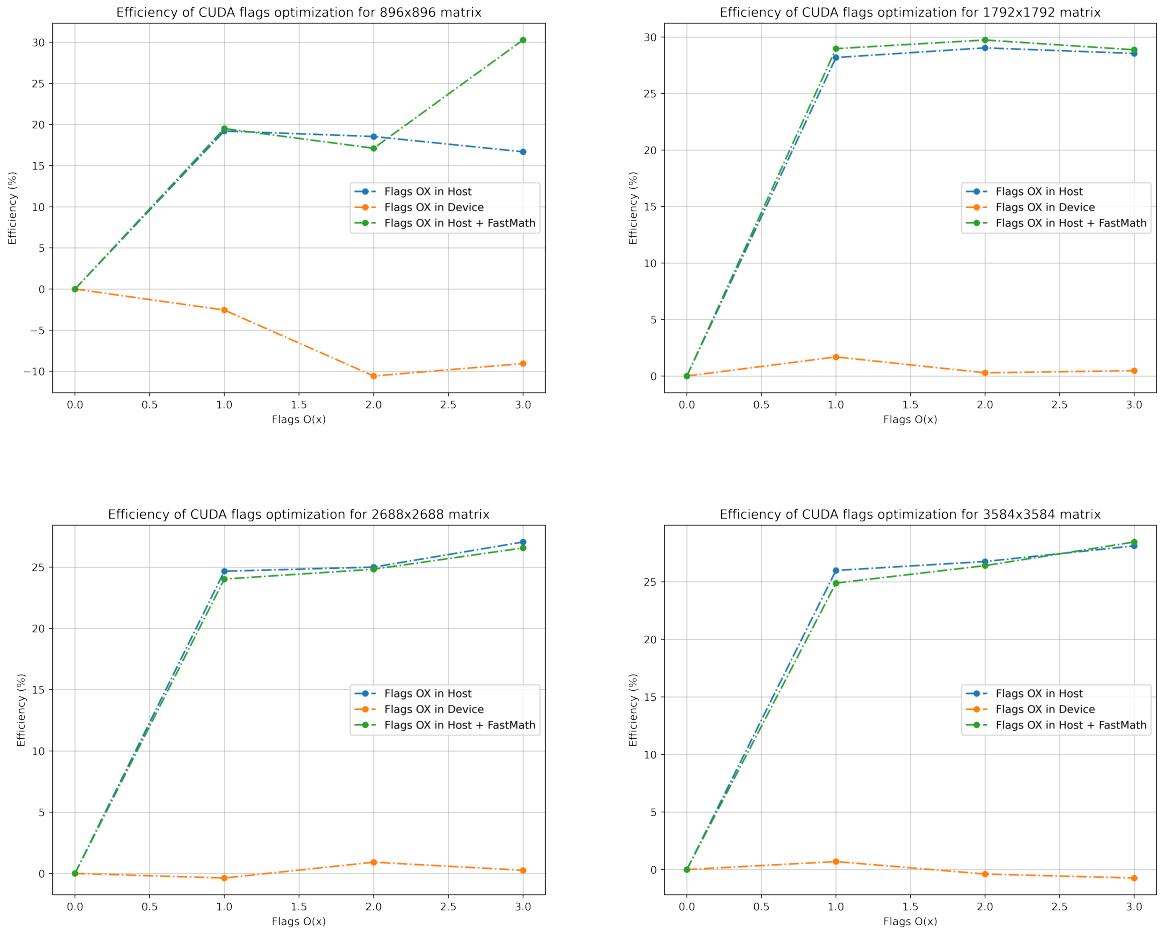


Figura 15 – Visualização da eficiência atingida em termos do tempo de execução em cada caso utilizando as flags de otimizações.

Onde novamente percebemos que não houve diferença substancial para o caso em que usamos a malha 896×896 . Extendendo, ainda, a analise para a métrica de speedup atingido em cada caso, através da equação (3.2), podemos percebemos com clareza que, no caso em que a malha utilizada é a malha 896×896 , não houve otimização ao utilizarmos as flags na compilação do programa.

O gráfico do speedup reforça a interpretação apresentada. Percebemos, portanto, que a medida que aumentamos a malha, as flags de otimizações se tornam importantes para garantir uma melhor perfomance do código.

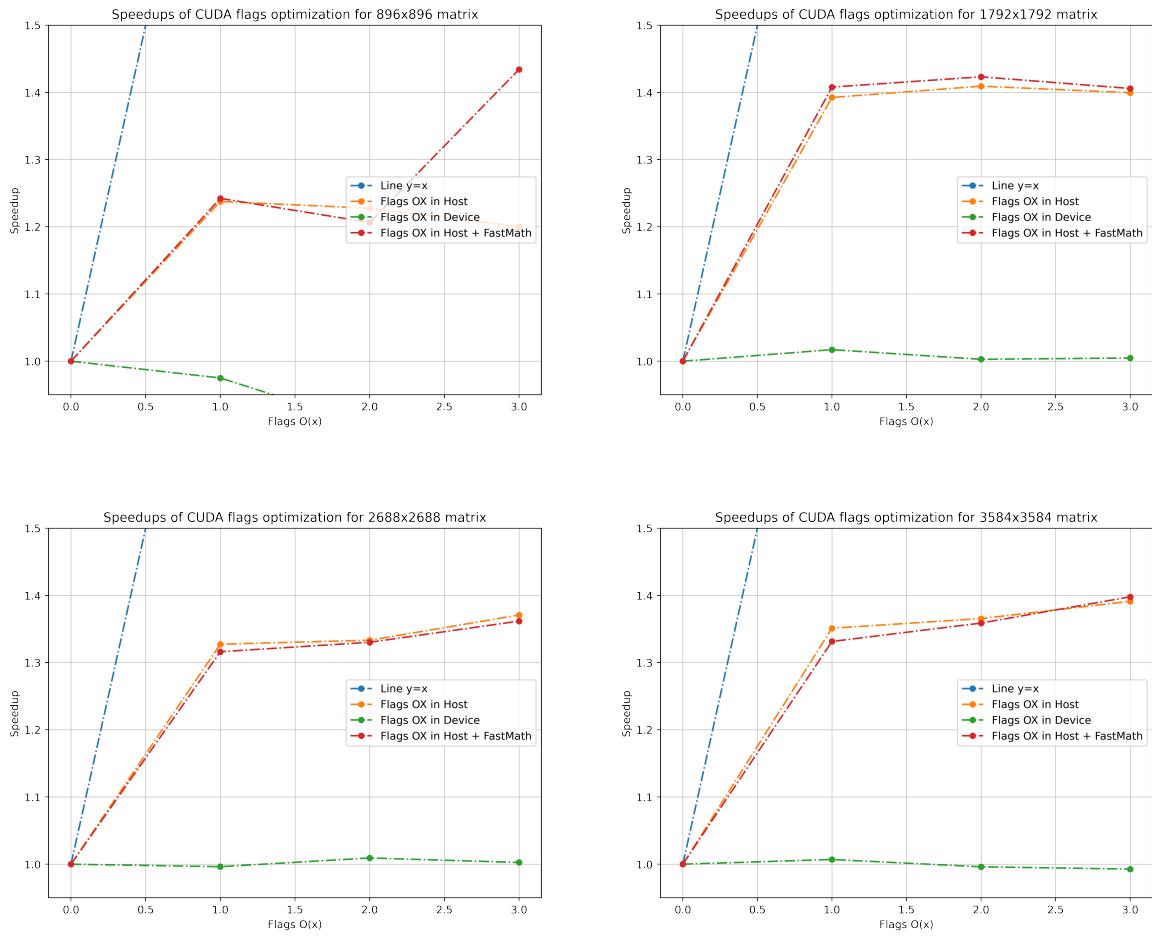


Figura 16 – Visualização do speedup obtido em termos de tempo de execução em cada caso utilizando as flags de otimizações.

Pensando em escalabilidade, a medida que aumentamos o tamanho da malha o tempo de execução é impactado pelas flags de otimização mas ocorre uma saturação de 30% de eficiência em termos do tempo da execução do programa, se comparado ao programa sem a utilização das flags.

Para finalizar a nossa análise, dispomos de um gráfico de barras mostrado na figura 17 para reforçar, ainda mais, nossa interpretação, contendo os melhores tempos obtidos para cada malha.

Repare que, em todo caso, as flags de otimizações `-OX -use_fast_math` foram as que melhor performaram no programa. Mais especificamente, a melhor flag utilizada se trata da flag `-O3 -use_fast_math`.

Comparando os dados obtidos para a melhor flag em relação ao programa em serial, temos os resultados dispostos na figura 18.

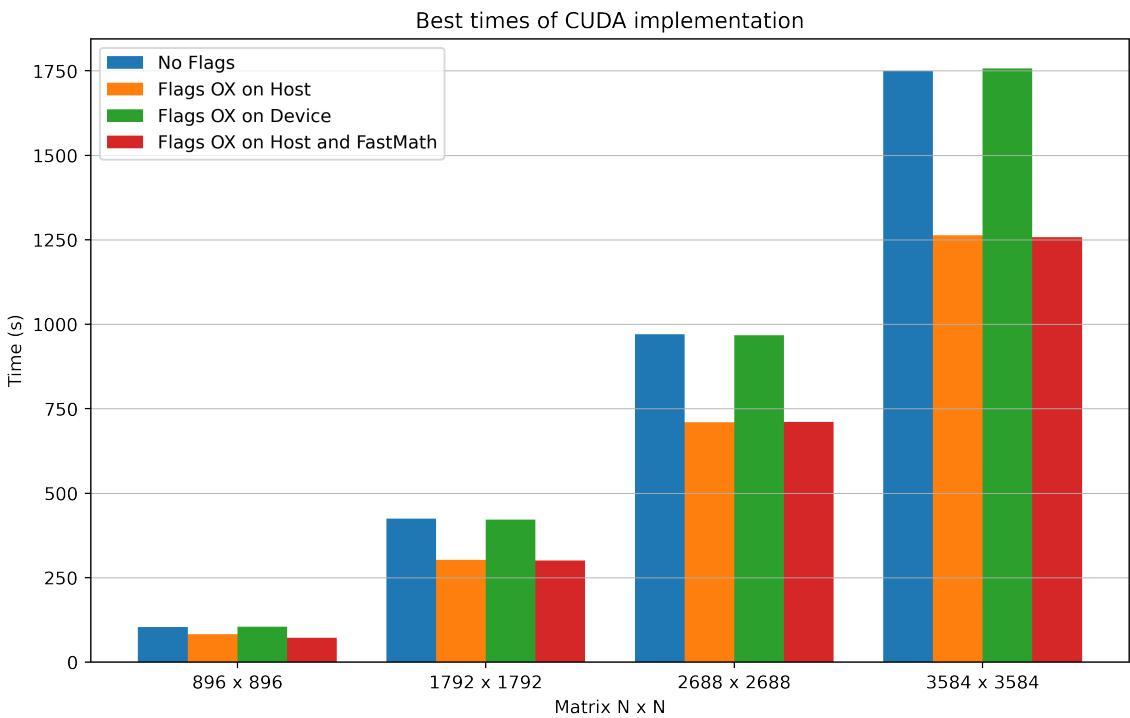


Figura 17 – Visualização dos melhores dados obtidos para o tempo de execução por malha utilizada.

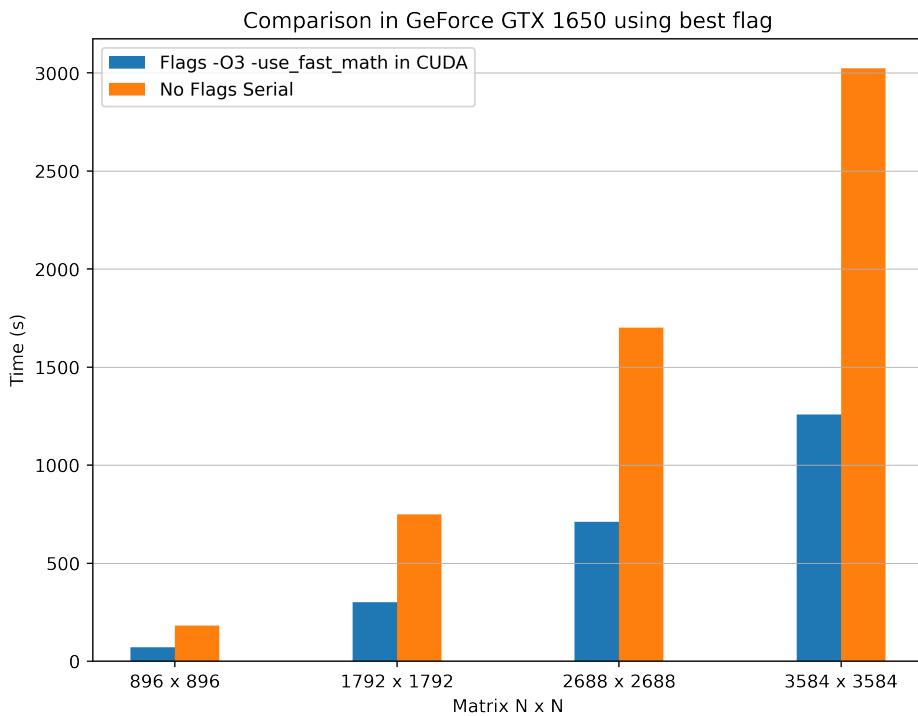


Figura 18 – Comparaçāo dos tempos de execuāo por malha para a melhor flag utilizada em relaāo ao programa serial.

A melhor métrica para encerrarmos a análise dos dados do benchmark se trata da métrica da eficiência da otimização, onde, usando a melhor flag contra o caso serial, obtemos:

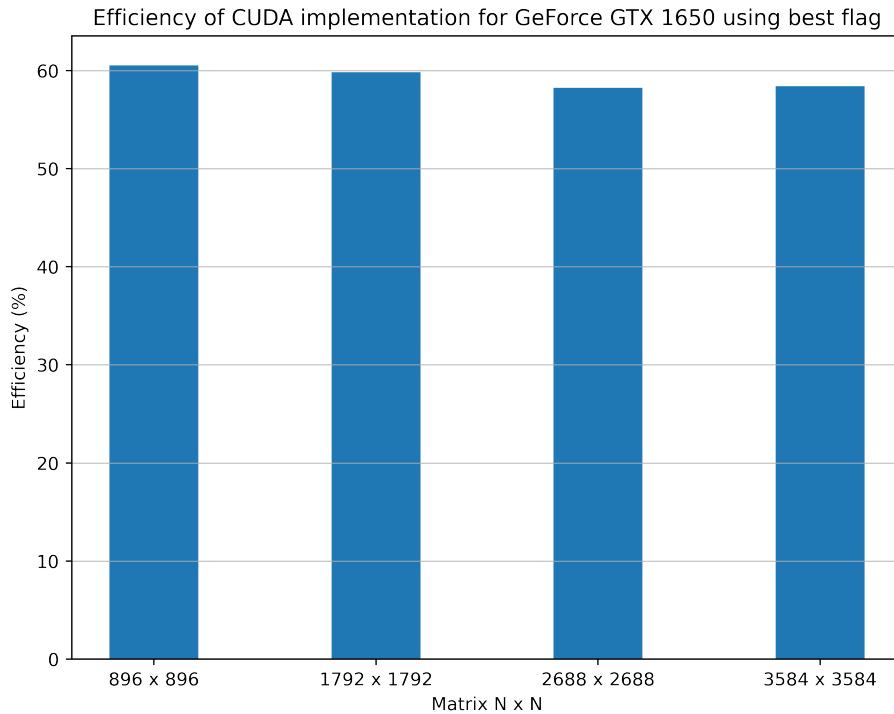


Figura 19 – Eficiência atingida por malha para a melhor flag utilizada em relação ao programa serial.

Tomando a média aritmética das eficiências atingidas, alcançamos a média de 60.4% de eficiência em relação ao tempo de execução do programa.

4.3.3 Benchmark [NVIDIA GeForce RTX 2060]

A melhor flag de otimização obtida na análise anterior se trata da flag de otimização -O3 -use_fast_math. A partir desse resultado, repetimos os cálculos da equação de onda para uma nova GPU, nesse caso, a GPGPU NVIDIA GeForce RTX 2060 que possui 1920 CUDA Cores e 12GB de memória VRAM.

Os resultados obtidos estão dispostos na figura 20 e demonstram um bom resultado de paralelização em relação ao serial.

Através desses resultados para o tempo de execução, traçamos as eficiências atingidas em cada caso. O resultado pode ser visualizado na figura 21, que sugere uma saturação de eficiência próxima de 60%.

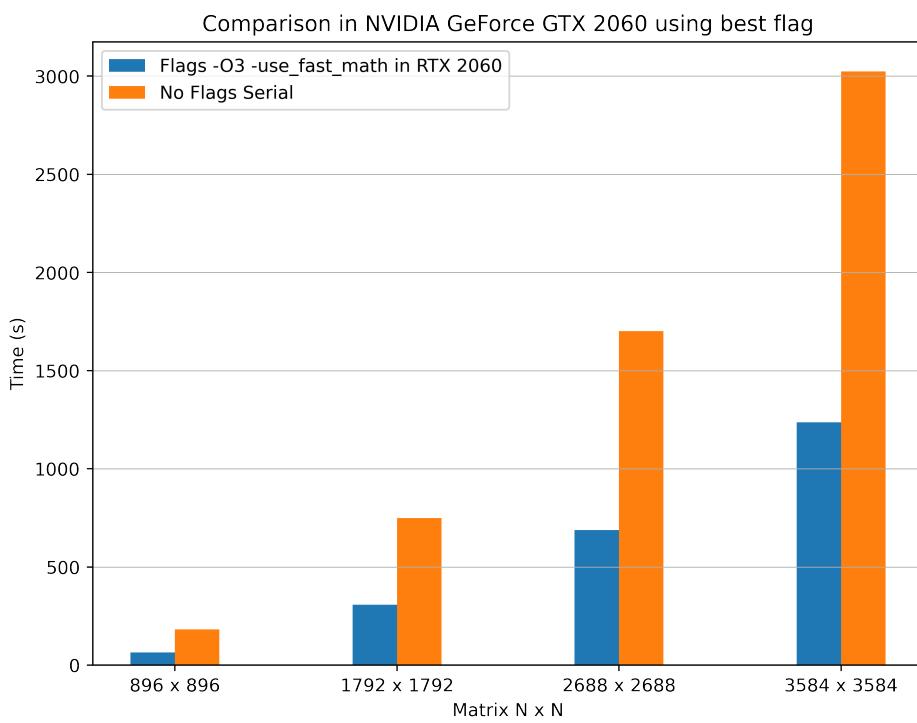


Figura 20 – Comparaçāo dos tempos de execuāo por malha utilizando a GPU NVIDIA GeForce RTX 2060 para a melhor flag utilizada em relaāo ao programa serial.

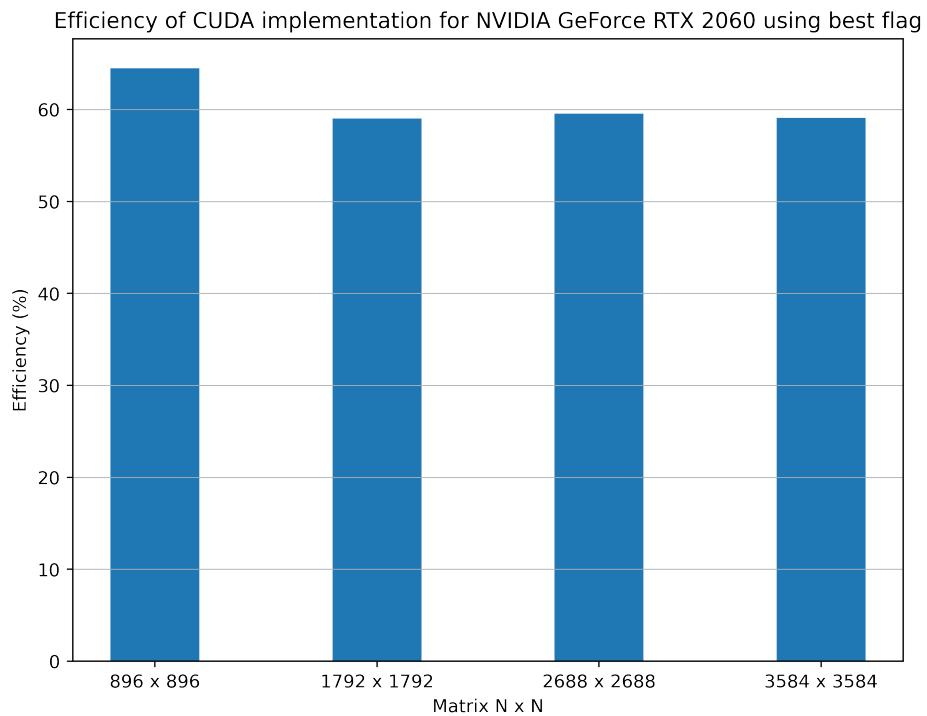


Figura 21 – Eficiēcia atingida por malha utilizando a GPU NVIDIA GeForce RTX 2060 para a melhor flag utilizada em relaāo ao programa serial.

4.3.4 Benchmark [NVIDIA Tesla K40]

Utilizando as mesmas configurações utilizadas anteriormente, repetimos as mesmas análises, agora usando a GPU NVIDIA Tesla K40, que possui 2880 CUDA Cores, onde obtemos os resultados para os tempos de execução dispostos na figura 22.

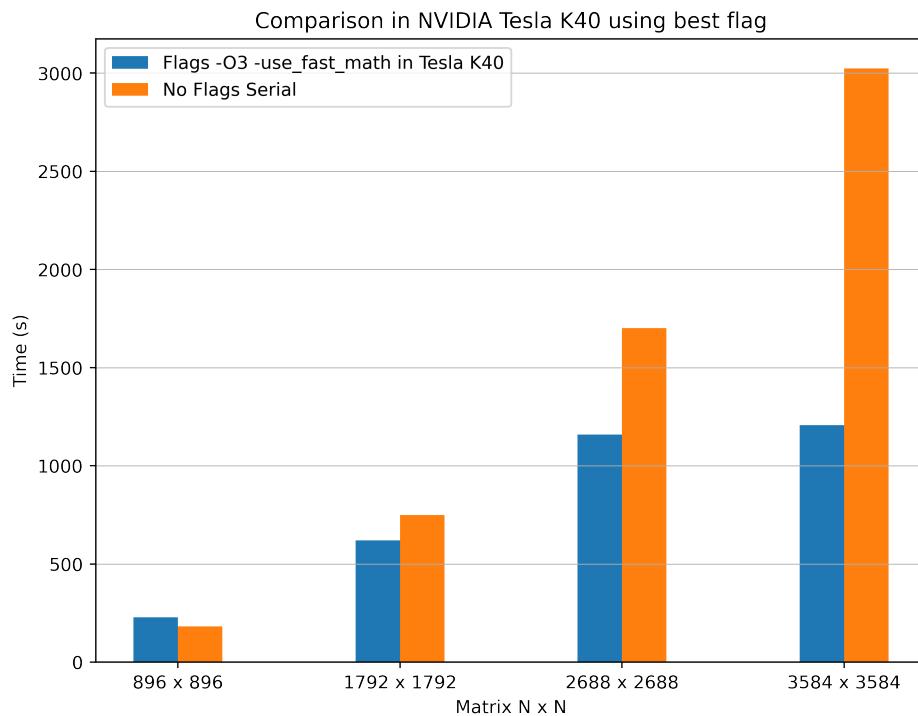


Figura 22 – Comparaçāo dos tempos de execuāo por malha utilizando a GPU NVIDIA Tesla K40 para a melhor flag utilizada em relaāo ao programa serial.

A partir desse resultado, foi possivel perceber que embora o nāmero de CUDA Cores da placa gráfica Tesla K40 fosse maior, para malhas pequenas, o resultado nāo foi satisfatório.

Tal resultado para malhas pequenas se deve ao fato de que, embora o nāmero de CUDA Cores da placa Tesla K40 seja maior, sua arquitetura e divisão de Streaming Multiprocessors (SM) é diferente da placa GTX 1650 apresentada anteriormente, resultando num maior gargalo computacional na divisão dos processos em cada SM. Aliado a isso, existe o gargalo da comunicação entre o HOST e o Device, tornando o tempo de execuāo maior ainda do que o do programa serial apresentado.

Podemos analisar essa informaāo mais claramente traçando o gráfico da eficiēcia obtida em relaāo ao programa serial utilizado, onde temos os seguintes resultados:

Repare que, conforme a malha se torna maior, a eficiēcia aumenta, atingindo 60% de eficiēcia para a malha de 3584×3584 . Isso acontece pois as divisões entre os SMs podem ser feitas de forma mais perfomática tornando a paralelizaāo mais eficiente.

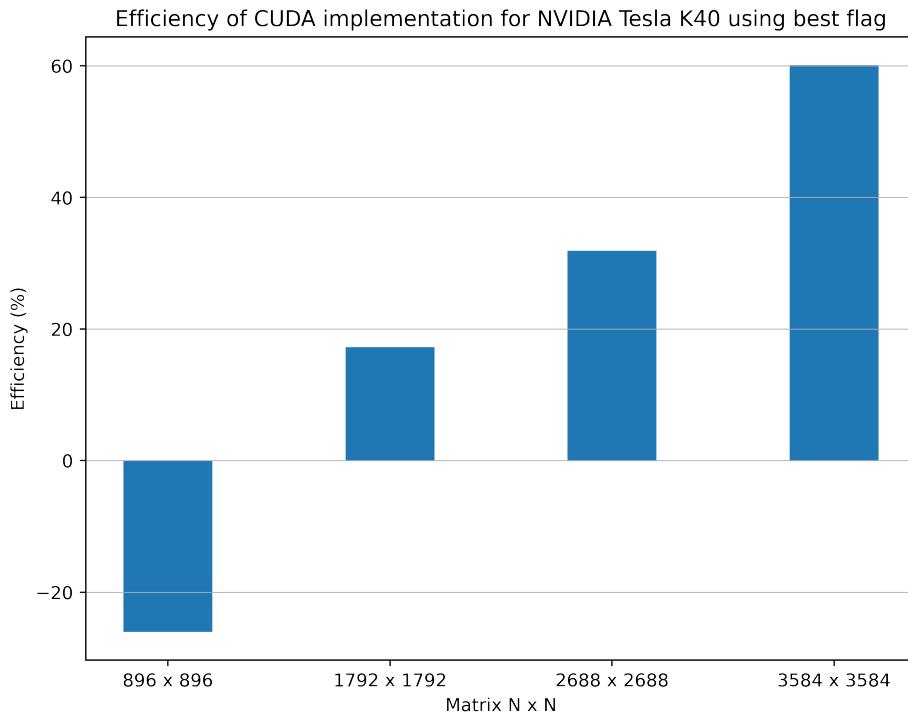


Figura 23 – Eficiência atingida por malha utilizando a GPU NVIDIA Tesla K40 para a melhor flag utilizada em relação ao programa serial.

4.3.5 Benchmark [NVIDIA Tesla V100]

Analogamente, o mesmo fora feito para a placa NVIDIA Tesla V100, sendo essa a placa mais poderosa utilizada nos benchmarks. Os resultados encontrados foram muito satisfatórios devido ao grande número de SMs e CUDA Cores disponibilizados, alcançando uma boa performance na paralelização.

Uma vez que a placa Tesla V100 possui muito mais poder computacional do que as outras placas gráficas analisadas, é possível perceber que a paralelização utilizando a V100 se torna a mais eficiente.

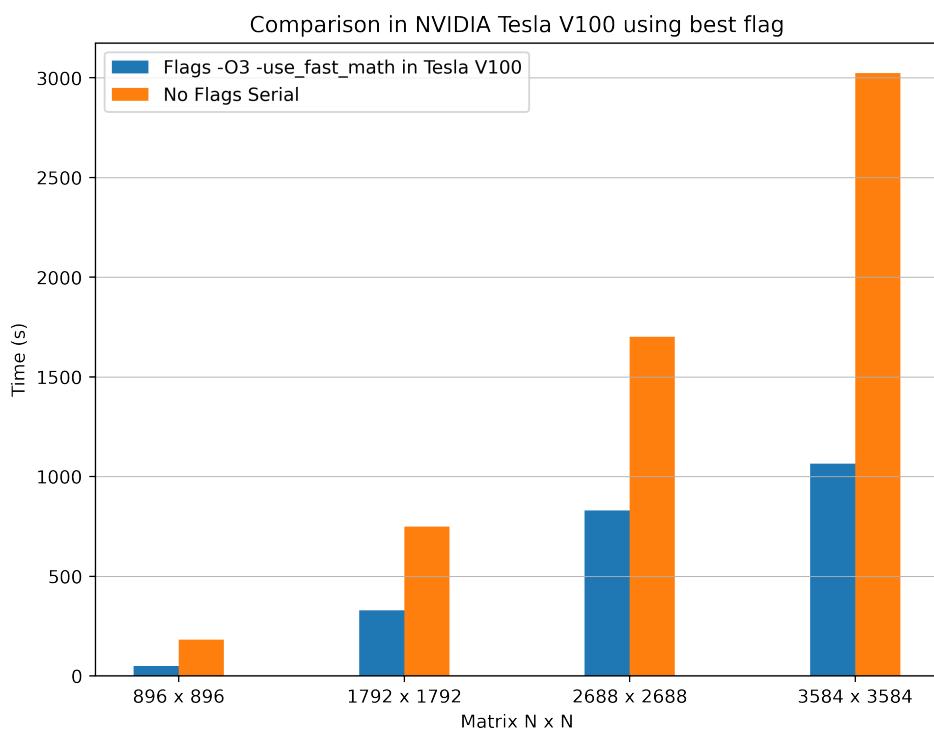


Figura 24 – Comparação dos tempos de execução por malha utilizando a GPU NVIDIA Tesla V100 para a melhor flag utilizada em relação ao programa serial.

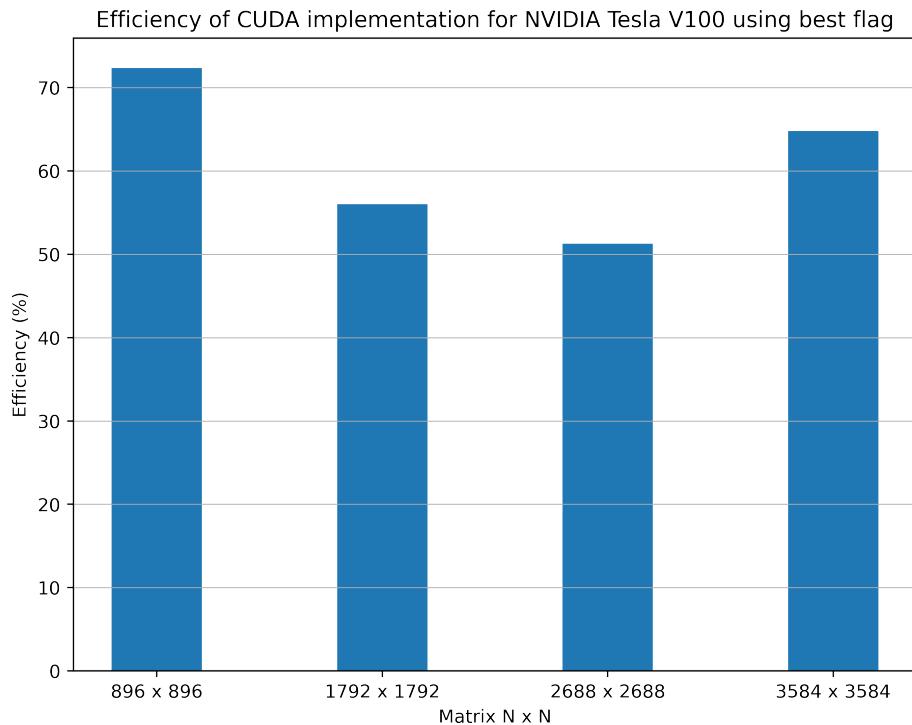


Figura 25 – Eficiência atingida por malha utilizando a GPU NVIDIA Tesla V100 para a melhor flag utilizada em relação ao programa serial.

Repare que ocorre uma saturação no aumento de eficiência da implementação, próximo de 72% de eficiência para a malha 896×896 . Isso acontece pelo fato de que, embora o poder computacional da placa seja muito grande, existe uma limitação na rede de transmissão de dados entre a GPU (device) e a CPU (host) que limita a performance da implementação.

Repare também que os resultados para a maior malha utilizada nos benchmarks, isto é, para a malha 3584×3584 , seguem os mesmos padrões das outras placas utilizadas no benchmark usando as melhores flags, que se trata da saturação próxima de 60%, o que não acontece para as malhas intermediárias.

5 Validação dos resultados

Foi implementado no algoritmo uma rotina para comparar os resultados numéricos com a solução analítica do problema, visando extrair os erros da solução numérica em comparação com a solução analítica do problema.

Os resultados encontrados estão dispostos na tabela a seguir:

Caso	Erro Percentual (%)
Serial	1.14
Best Flag Serial	7.19
GeForce GTX 1650 + Best Flag CUDA	4.96
Tesla K40 + Best Flag CUDA	2.74
Tesla V100 + Best Flag CUDA	1.53

Repare que o caso em que foi usado a melhor flag para o programa em serial foi o caso em que houve o maior erro percentual em relação à solução analítica. Isso pode ser explicado levando em conta o fato de que as flags de otimização utilizadas no modo serial tornam a compilação mais agressiva, podendo ocasionar erros de arredondamento em situações em que precisamos de maior uma maior precisão dupla.

Repare também que a GPGPU NVIDIA Tesla V100, dentre as placas utilizadas para a implementação do CUDA, foi a que obteve menor erro percentual, devido ao fato de sua grande precisão dupla.

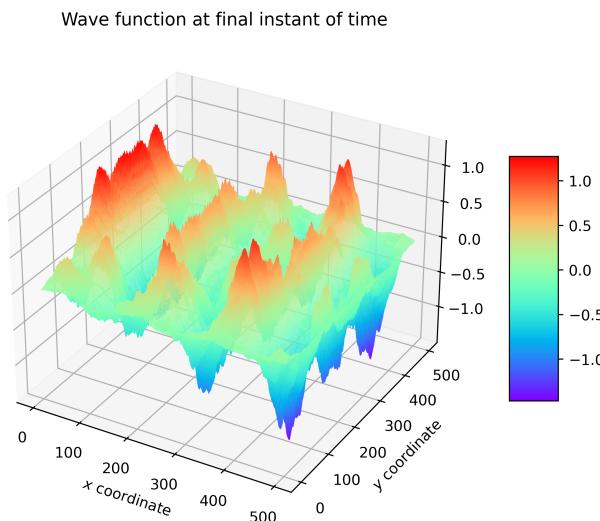


Figura 26 – Onda calculada no instante final estabelecido. Resultado do cálculo da equação de onda utilizando a GPU NVIDIA Geforce GTX 1650.

6 Conclusões

Os processos de paralelização são uma ferramenta computacional muito utilizada para extensos cálculos e métodos que requerem um poder computacional mais distribuído e otimizado do que propriamente na natureza serial que encontramos habitualmente.

Em termos físicos, em que estamos interessados na modelagem de fenômenos da natureza, percebemos que muitos dos processos que requerem explicitações requerem uma alta gama de dados e nuances que as fórmulas fornecem em pequenos intervalos. Por conta disso, a visualização e a exploração do caráter teórico da Física é, muitas vezes, limitado pela possibilidade de se realizar esses processos e em alguns não possuindo solução analítica.

Buscamos ao longo desse trabalho reforçar a importância do esforço pela busca por soluções paralelas para problemas já bem estabelecidos dentro da física, como foi o caso da equação de onda, visto que a solução com base no programa serial consumia bastante tempo do usuário.

Através da API CUDA, conseguimos parallelizar o código utilizando placas de vídeo (GPGPUs) e pudemos analisar a eficiência de diferentes placas, sendo elas GPUs para uso pessoal (GeForce GTX 1650) e GPUs de uso profissional (Tesla K40 e Tesla V100), onde obtivemos bons resultados.

Diversos avanços estão sendo feitos em relação à programação científica de alto desempenho, dessa forma, a utilização de APIs como CUDA se torna vital para que o usuário possa extrair o máximo de performance da tecnologia atual disponível.

Referências

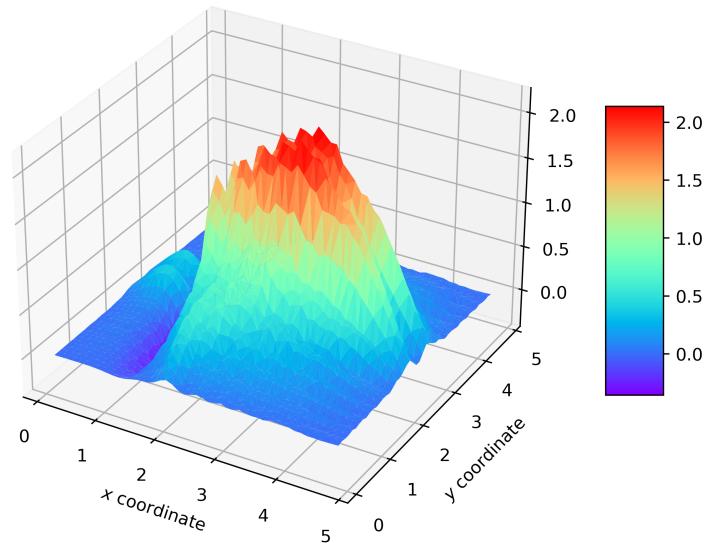
- [1] NVIDIA CUDA Home Page. <https://developer.nvidia.com/cuda-zone>. Accessed: 2022-06-26. Citado na página 16.
- [2] TECHNICAL BLOG: How to Optimize Data Transfers in CUDA C/C++. <https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/>. Accessed: 2022-06-27. Citado na página 20.

7 Apêndices

7.1 Apêndice A - Visualização da solução para diferentes malhas e condições iniciais

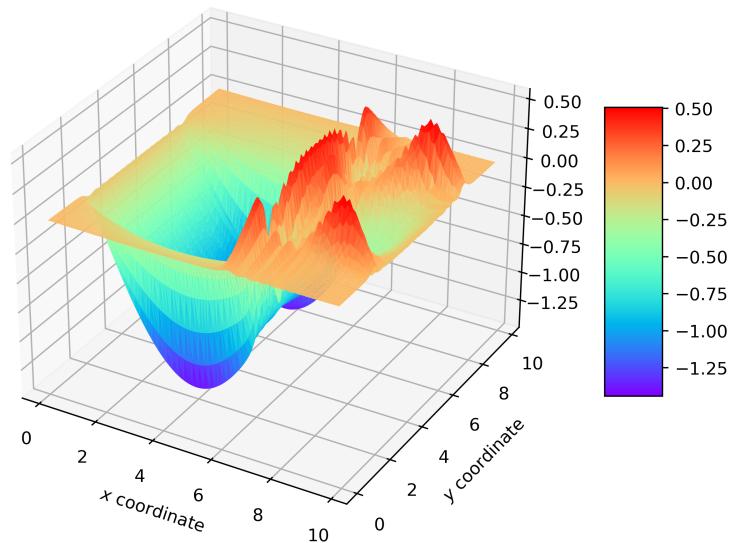
7.1.1 A.1 - Malha 5×5 e $N = 100$

Wave function at final instant of time



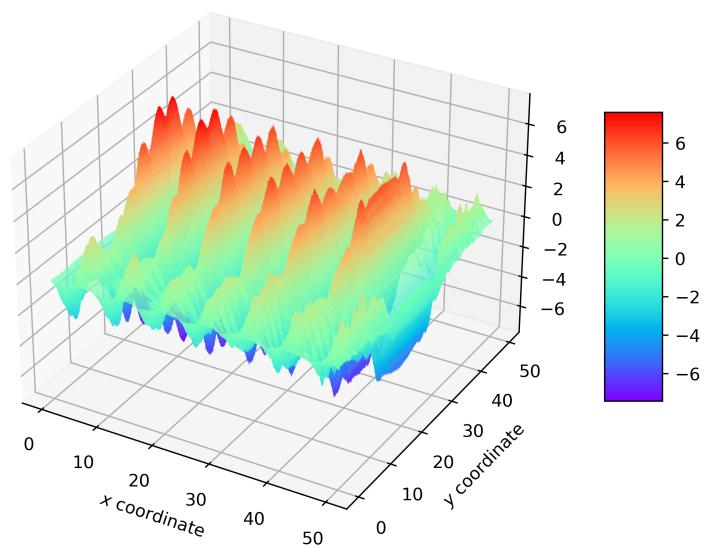
7.1.2 A.2 - Malha 10×10 e $N = 100$

Wave function at final instant of time



7.1.3 A.3 - Malha 50×50 e $N = 1000$

Wave function at final instant of time



8 Anexos

8.1 A - Código serial

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <math.h>
5 #include <time.h>
6
7 #define N 1000
8 #define xInicial 0
9 #define xFinal 500.0
10 #define yInicial 0
11 #define yFinal 500.0
12 #define finalTime 1000
13 #define alpha 0.4
14 #define gamma 0.8
15
16 double ***allocArray()
17 {
18
19     int i, j;
20     double ***waveFunction;
21
22     waveFunction = (double ***)malloc(N * sizeof(double));
23     for (i = 0; i < N; i++)
24     {
25
26         waveFunction[i] = (double **)malloc(N * sizeof(double));
27
28         for (j = 0; j < N; j++)
29         {
30
31             waveFunction[i][j] = (double *)malloc(3 * sizeof(double));
32         }
33     }
34
35     return waveFunction;
36 }
37
38 void ***initialCondition(double ***wave, double dx, double dy)
39 {
40 }
```

```

41     int i , j ;
42
43     for (i = 0; i < N; i++)
44     {
45         for (j = 0; j < N; j++)
46         {
47
48             wave[i][j][0] = 4 * sin(M_PI * i / 75);
49         }
50     }
51 }
52
53 void ***contourCondition(double ***wave, double dx, double dy)
54 {
55
56     int i , j , t ;
57
58     for (j = 0; j < N; j++)
59     {
60         for (t = 0; t < 3; t++)
61         {
62             wave[0][j][t] = 0;
63             wave[N - 1][j][t] = 0;
64         }
65     }
66
67     for (i = 0; i < N; i++)
68     {
69         for (t = 0; t < 3; t++)
70         {
71             wave[i][0][t] = 0;
72             wave[i][N - 1][t] = 0;
73         }
74     }
75 }
76
77 void ***derivativeCondition(double ***wave, double dx, double dy)
78 {
79
80     int i , j ;
81
82     for (i = 1; i < N - 1; i++)
83     {
84         for (j = 1; j < N - 1; j++)
85         {
86             wave[i][j][1] = (2 * wave[i][j][0] * (1 - alpha * alpha -
gamma * gamma) + alpha * alpha * wave[i + 1][j][0] + alpha * alpha *

```

```

        wave[i - 1][j][0] + gamma * gamma * wave[i][j + 1][0] + gamma * gamma
        * wave[i][j - 1][0]) / 2;
    87    }
    88 }
89 }
90
91 void ***finiteDifference(double ***wave, double dx, double dy)
92 {
93
94     int i, j, t;
95
96     for (t = 1; t < finalTime; t++)
97     {
98         for (i = 1; i < N - 1; i++)
99         {
100            for (j = 1; j < N - 1; j++)
101            {
102                wave[i][j][(t + 1) % 3] = 2 * wave[i][j][t % 3] * (1 -
103 alpha * alpha - gamma * gamma) - wave[i][j][(t - 1) % 3] + alpha *
104 alpha * wave[i + 1][j][(t % 3)] + alpha * alpha * wave[i - 1][j][t %
105 3] + gamma * gamma * wave[i][j + 1][t % 3] + gamma * gamma * wave[i][
106 j - 1][t % 3];
107            }
108        }
109    }
110 }
111
112 void writeFiles(double ***wave, double dx, double dy)
113 {
114
115     int i, j, t;
116     FILE *fileDynamicPlot, *fileStaticPlot;
117
118 // fileDynamicPlot = fopen("Wave.dat", "w");
119 fileStaticPlot = fopen("WaveStatic.dat", "w");
120
121 // fprintf(fileDynamicPlot, "x\ty\tt\tf\n");
122 fprintf(fileStaticPlot, "x\ty\tt\tf\n");
123
124 // for (t = 1; t < finalTime; t++) {
125 //   for (i = 1; i < N-1; i++) {
126 //     for (j = 1; j < N-1; j++) {
127 //       fprintf(fileDynamicPlot, "%lf\t%lf\t%d\t%lf\n", i*dx,
128 j*dy, t, wave[i][j][(t+1)%3]);
129 //     }
130 //   }
131 // }

```

```

127
128     for (i = 1; i < N - 1; i++)
129     {
130         for (j = 1; j < N - 1; j++)
131         {
132             fprintf(fileStaticPlot, "%lf\t%lf\t%d\t%lf\n", i * dx, j *
133 dy, finalTime - 1, wave[i][j][(finalTime) % 3]);
134         }
135     }
136
137 // fclose(fileDynamicPlot);
138 fclose(fileStaticPlot);
139
140 void actionWork(double ***wave)
141 {
142
143     double dx, dy, dt;
144
145     dx = (xFinal - xInicial) / N;
146     dy = (yFinal - yInicial) / N;
147     printf("dx: %lf\tdy: %lf\n", dx, dy);
148     printf("Malha: %d x %d\n", N, N);
149     printf("Tempo total: %d\n", finalTime);
150
151     printf("Colocando condição inicial.\n");
152     initialCondition(wave, dx, dy);
153
154     printf("Colocando condição de contorno.\n");
155     contourCondition(wave, dx, dy);
156
157     printf("Colocando condição da derivada.\n");
158     derivativeCondition(wave, dx, dy);
159
160     printf("Iniciando cálculo da função de onda.\n");
161     finiteDifference(wave, dx, dy);
162
163     printf("Escrevendo no arquivo\n");
164     writeFiles(wave, dx, dy);
165 }
166
167 void main()
168 {
169
170     double ***waveFunction;
171
172     system("clear");

```

```
173     clock_t beginTime = clock();
174
175     printf("Alocando a memoria do array.\n");
176     waveFunction = allocArray();
177
178     printf("Comeando os calculos.\n");
179     actionWork(waveFunction);
180
181     clock_t endTime = clock();
182
183     printf("Time: %10.2f seconds \n", (endTime - beginTime) / (1.0 *
184     CLOCKS_PER_SEC));
184 }
```

Listing 8.1 – Programa Serial

8.2 B - Implementação da API CUDA

```
1 #include "cuda_runtime.h"
2 #include "device_launch_parameters.h"
3 #include <malloc.h>
4 #include <stdio.h>
5 #include <math.h>
6 #include <time.h>
7
8
9 #define mu 0.01
10 #define T 40
11 #define N 1000
12 #define tempoTotal 1000
13 #define xInicial 0
14 #define xFinal 500.0
15 #define yInicial 0
16 #define yFinal 500.0
17 #define alpha 0.4
18 #define gamma 0.8
19
20 #define CHECK(call) \
21     { \
22         cudaError_t error = call; \
23         if (error != cudaSuccess) \
24             { \
25                 fprintf(stderr, "Error: %s:%d, ", __FILE__, __LINE__); \
26                 fprintf(stderr, "code: %d, reason: %s\n", error, \
27                         cudaGetErrorString(error)); \
28             } \
29     } \
30
31 __global__ void finiteDifferenceKernel(double *wave, double *waveFuture,
32                                         double *wavePast)
33 {
34     unsigned int i = blockIdx.x;
35     unsigned int j = threadIdx.x;
36
37     if ((i > 0 && i < N - 1) && (j > 0 && j < N - 1))
38         waveFuture[i * blockDim.x + j] = 2 * wave[i * blockDim.x + j] *
39             (1 - alpha * alpha - gamma * gamma) - wavePast[i * blockDim.x + j] +
40             alpha * alpha * wave[(i + 1) * blockDim.x + j] + alpha * alpha * wave
41             [(i - 1) * blockDim.x + j] + gamma * gamma * wave[i * blockDim.x + (j
42             + 1)] + gamma * gamma * wave[i * blockDim.x + (j - 1)];
43 }
44
45 void writeFiles(double *wave, double dx, double dy)
46 {
```

```

42     int i , j;
43     FILE *fileStaticPlot;
44
45     fileStaticPlot = fopen("WaveStatic2.dat" , "w");
46
47     fprintf(fileStaticPlot , "x\ty\tf\n");
48
49     for (i = 1; i < N - 1; i++)
50     {
51         for (j = 1; j < N - 1; j++)
52         {
53             fprintf(fileStaticPlot , "%lf\t%lf\t%lf\n" , i * dx , j * dy ,
54             wave[i * N + j]);
55         }
56     }
57
58     fclose(fileStaticPlot);
59 }
60
61 void initialCondition(double *wave)
62 {
63     int i , j;
64     for (j = 0; j < N; j++)
65     {
66         for (i = 0; i < N; i++)
67         {
68             wave[i * N + j] = 4 * sin(M_PI * i / 75.0);
69         }
70     }
71 }
72
73 void derivativeCondition(double *wave , double *wavePast)
74 {
75     int i , j;
76
77     for (i = 1; i < N - 1; i++)
78     {
79         for (j = 1; j < N - 1; j++)
80         {
81             wave[i * N + j] = (2 * wavePast[i * N + j] * (1 - alpha *
82             alpha - gamma * gamma) + alpha * alpha * wavePast[(i + 1) * N + j] +
83             alpha * alpha * wavePast[(i - 1) * N + j] + gamma * gamma * wavePast[
84             i * N + (j + 1)] + gamma * gamma * wavePast[i * N + (j - 1)]) / 2;
85         }
86     }
87 }
```

```

85
86 double resultsValidation(double *wave, double dx, double dy)
87 {
88
89     int i, j;
90     double erro, maiorErro, analitica;
91
92     for (i = 0; i < N; i++)
93     {
94         for (j = 0; j < N; j++)
95         {
96
97             analitica = sin((j * M_PI * dx) / xFinal) * sin((i * M_PI *
98             dy) / yFinal) * ((-2000 / ((M_PI) * (M_PI))) * ((sin(M_PI * j) * (cos(
99             M_PI * i) - 1)) / ((j * j - 250000) * i))) * cos(sqrt(mu / T) * (
100                sqrt((j * M_PI / xFinal) * (j * M_PI / xFinal) + (i * M_PI / yFinal)
101                * (i * M_PI / yFinal))) * tempoTotal);
102
103             erro = fabs((wave[i * N + j] - analitica) / analitica);
104
105             if (i == 0 && j == 0)
106             {
107                 maiorErro = erro;
108             }
109             else
110             {
111                 if (erro > maiorErro)
112                 {
113                     maiorErro = erro;
114                 }
115             }
116         }
117     }
118     return maiorErro;
119 }
120
121 void deviceCapabilities()
122 {
123     cudaDeviceProp prop;
124     int count;
125     cudaGetDeviceCount(&count);
126     for (int i = 0; i < count; i++)
127     {
128         cudaGetDeviceProperties(&prop, i);
129         printf("\n --- General Information for device %d ---\n", i);

```

```

128     printf("Name: %s\n", prop.name);
129     printf("Compute capability: %d.%d\n", prop.major, prop.minor);
130     printf("Clock rate: %d\n", prop.clockRate);
131     printf("Device copy overlap: ");
132     if (prop.deviceOverlap)
133         printf("Enabled\n");
134     else
135         printf("Disabled\n");
136     printf("Kernel execition timeout : ");
137     if (prop.kernelExecTimeoutEnabled)
138         printf("Enabled\n");
139     else
140         printf("Disabled\n");
141     printf("\n --- Memory Information for device %d ---\n", i);
142     printf("Total global mem: %ld\n", prop.totalGlobalMem);
143     printf("Total constant Mem: %ld\n", prop.totalConstMem);
144     printf("Max mem pitch: %ld\n", prop.memPitch);
145     printf("Texture Alignment: %ld\n", prop.textureAlignment);
146     printf("\n --- MP Information for device %d ---\n", i);
147     printf("Multiprocessor count: %d\n",
148            prop.multiProcessorCount);
149     printf("Shared mem per mp: %ld\n", prop.sharedMemPerBlock);
150     printf("Registers per mp: %d\n", prop.regsPerBlock);
151     printf("Threads in warp: %d\n", prop.warpSize);
152     printf("Max threads per block: %d\n",
153            prop.maxThreadsPerBlock);
154     printf("Max thread dimensions: (%d, %d, %d)\n",
155            prop.maxThreadsDim[0], prop.maxThreadsDim[1],
156            prop.maxThreadsDim[2]);
157     printf("Max grid dimensions: (%d, %d, %d)\n",
158            prop.maxGridSize[0], prop.maxGridSize[1],
159            prop.maxGridSize[2]);
160     printf("\n\n");
161 }
162 }
163
164 void actionWork(double dx, double dy)
165 {
166
167     int i, j, t;
168
169     double *hostWave, *hostWaveFuture, *hostWavePast;           // Host
170     variables
171     double *deviceWave, *deviceWaveFuture, *deviceWavePast; // Device
172     Variables
173     double erro;

```

```

173     printf("Alocando memoria no host\n");
174     hostWave = (double *)calloc((N * N), sizeof(double));
175     hostWaveFuture = (double *)calloc((N * N), sizeof(double));
176     hostWavePast = (double *)calloc((N * N), sizeof(double));
177
178     printf("Colocando condi o inicial.\n");
179     initialCondition(hostWavePast);
180     printf("Colocando condi o da derivada.\n");
181     derivativeCondition(hostWave, hostWavePast);
182
183     printf("Alocando memoria no Device\n");
184     CHECK(cudaMalloc(&deviceWave, (N * N) * sizeof(double)));
185     CHECK(cudaMalloc(&deviceWaveFuture, (N * N) * sizeof(double)));
186     CHECK(cudaMalloc(&deviceWavePast, (N * N) * sizeof(double)));
187
188     printf("Iniciando calculo da fun o de onda.\n");
189     for (t = 0; t < tempoTotal; t++)
190     {
191         CHECK(cudaMemcpy(deviceWave, hostWave, (N * N) * sizeof(double),
192                         cudaMemcpyHostToDevice));
193         CHECK(cudaMemcpy(deviceWaveFuture, hostWaveFuture, (N * N) *
194                     sizeof(double), cudaMemcpyHostToDevice));
195         CHECK(cudaMemcpy(deviceWavePast, hostWavePast, (N * N) * sizeof(
196                     double), cudaMemcpyHostToDevice));
197
198         finiteDifferenceKernel<<<N, N>>>(deviceWave, deviceWaveFuture,
199         deviceWavePast);
200
201         CHECK(cudaMemcpy(hostWave, deviceWave, (N * N) * sizeof(double),
202                         cudaMemcpyDeviceToHost));
203         CHECK(cudaMemcpy(hostWaveFuture, deviceWaveFuture, (N * N) *
204                     sizeof(double), cudaMemcpyDeviceToHost));
205         CHECK(cudaMemcpy(hostWavePast, deviceWavePast, (N * N) * sizeof(
206                     double), cudaMemcpyDeviceToHost));
207
208         for (i = 1; i < N - 1; i++)
209         {
210             for (j = 1; j < N - 1; j++)
211             {
212                 hostWavePast[i * N + j] = hostWave[i * N + j];
213                 hostWave[i * N + j] = hostWaveFuture[i * N + j];
214             }
215         }
216
217         printf("Escrevendo no arquivo o resultado do c lculo\n");
218         writeFiles(hostWave, dx, dy);

```

```

213
214     erro = resultsValidation(hostWave, dx, dy);
215
216     printf("Erro da solução numérica: %lf\n", erro);
217
218     printf("Liberando memória no host e device ... \n");
219
220     free(hostWave);
221     free(hostWaveFuture);
222     free(hostWavePast);
223
224     cudaFree(deviceWave);
225     cudaFree(deviceWaveFuture);
226     cudaFree(deviceWavePast);
227 }
228
229 int main()
230 {
231     double dx, dy;
232
233     dx = (xFinal - xInicial) / N;
234     dy = (yFinal - yInicial) / N;
235
236     deviceCapabilities();
237
238     printf("dx: %lf\tdy: %lf\n", dx, dy);
239     printf("Malha: %d x %d\n", N, N);
240     printf("Tempo total: %d\n", tempoTotal);
241
242     clock_t beginTime = clock();
243
244     actionWork(dx, dy);
245
246     clock_t endTime = clock();
247
248     printf("Time: %10.2f seconds \n", (endTime - beginTime) / (1.0 *
249     CLOCKS_PER_SEC));
250     return 0;
251 }
```

Listing 8.2 – Programa Serial