MAC0219 - Relatório EP02

Nathália Yukimi Uchiyama Tsuno (14600541)
Renata Meyer Hobold (14605790)

24 de Novembro de 2024

1 Introdução

Neste trabalho, o objetivo principal é explorar a resolução da equação de calor em estado estacionário, um problema fundamental em diversas áreas da ciência e engenharia, como a física térmica e a engenharia de materiais. A equação de calor descreve a distribuição de temperatura em um meio, e no contexto deste EP, utilizamos a equação de Laplace para modelar a propagação de calor em um espaço bidimensional com condições de contorno fixas. A partir de um código sequencial já fornecido, buscamos paralelizar o cálculo da distribuição de temperatura utilizando CUDA.

A resolução do problema se dá por meio do método de diferenças finitas, onde a área de interesse é discretizada em uma malha de pontos, e a temperatura em cada ponto interno é calculada como uma média das temperaturas nos pontos adjacentes. O processo iterativo converge até que a distribuição de temperatura alcance um valor estável, permitindo a obtenção de uma solução numérica aproximada para o problema proposto. A implementação sequencial desse método apresenta bons resultados, mas ao paralelizar a execução usando CUDA, buscamos uma aceleração do processo.

Neste relatório, abordamos a implementação do código paralelizado, discutindo os desafios enfrentados, as estratégias adotadas para otimizar a execução e os resultados obtidos, comparando o desempenho do código sequencial com a versão paralelizada.

À princípio, considere que todos os experimentos foram executados na máquina `Vegeta`, pertencente à `Rede Linux`, com as seguintes especificações:

CPU	AMD Ryzen 7 5700G with Radeon Graphics
RAM	125Gi
SO	Debian GNU/Linux 12 (bookworm)
Placa de Vídeo	NVIDIA GeForce RTX 3060

Table 1: Especificações da Rede Linux

A versão atual é o CUDA Toolkit 11.8. E a versão atual do `nvcc` é 11.8.89.

2 Notas de Compilação

O `MAKEFILE` está configurado para o `CUDAPATH` e o `NVCC` setados conforme na `Rede Linux`. Portanto, se o uso não for lá, sugerimos modificar esses caminhos antes de executar o `MAKEFILE`.

Ademais, para executar ambos os programas, é necessário executar

```
make heat-tarefa1
make heat-tarefa2
make heat-tarefa3
```

3 Tarefa 1

A tarefa 1, por engano, já foi enviada feita. Não há nada a fazer.

4 Tarefa 2

4.1 Implementação de uma Subrotina Apenas no `Host`

No programa devidamente descrito em `heat-tarefa2.cu`, é possível constatar a implementação da rotina que calcula apenas a distribuição de calor apenas no `host`. Trata-se da `jacobi_iteration`, uma subrotina que não retorna nada e que recebe um ponteiro para um array `h`, outro para `g`, o tamanho dos arrays e a quantidade máxima de iterações desejadas.

```
void jacobi_iteration(double *h, double *g, int n, int iter_limit)
```

Na implementação da `jacobi_iteration`, utilizam-se arrays e não matrizes, para melhor comparação com a versão produzida pela GPU. Verifique que a manipulação sobre estrtuturas 1D são mais eficientes e admitidas, por padrão, pelo CUDA. Assim, tal que descrito na implementação para GPU, obtém-se uma transformação bijetora entre as posições de memórias a serem acesadas.

O funcionamento da `jacobi_iteration` se dá essencialmente como, ao longo das iterações, vão se calculando, para cada i-ésima posição do array `g`, o quarto da soma de seus vizinhos (numa interpretação matricial). E, ao fim de cada t-ésima iteração, atualiza-se o array `h` com os valores descritos em `g`.

Adicionalmente, é suposto que `h` já esteja devidamente setado com os valores de inicialização.

4.2 Implementação de uma Subrotina `Kernel`

Como queríamos paralelizar o código, nós criamos um `kernel` — uma rotina feita para ser executados em paralelo por diversas threads — que escreve em `g` o novo valor de um pixel específico (a depender do ID da thread executando-o e do bloco a qual essa thread pertence) a partir das informações em `h`.

```
__global__ void jacobi_kernel(double *h, double *g, int n)
```

Como cada chamada escreve necessariamente em píxeis distintos da imagem, então não existe condição de corrida, desde que todos eles estejam na mesma iteração. Isso é responsabilidade da `main`, que sincroniza todas as threads de uma iteração antes de realizar a próxima.

Também passa-se como parâmetro o tamanho da imagem `n`, porque as bordas são intocáveis; a função realiza a verificação de se o píxel-alvo não é uma borda antes de tentar alterá-la, e se for, deixa-a do jeito que está. Isso garante que as bordas estejam consistentemente com a mesma temperatura durante todas as iterações.

4.3 Implementação de uma Rotina Comparativa entre as Versões da CPU e da GPU

Afim de dar credibilidade e confiabilidade no código descrito e devidamente direcionado à execução na GPU, implementou-se uma rotina comparativa entre a versão sequencial, mas, com a corretude atestada, oriunda da CPU, e a versão de menor complexidade, oriunda da GPU. Trata-se da `compareResults`, uma rotina que não retorna nada e que recebe um ponteiro para um array `C1`, um ponteiro para o array `C2` e o tamanho das instâncias.

```
void compareResults(double *C1, double *C2, int N)
```

Na implementação de `compareResults`, tem-se setado, por padrão, um valor `epsilon`, correspondente a 2×10^{-2}, que indica a margem máxima de erro entre cada i-ésima posição acessada dos arrays `C1` e `C2` a serem comparados. Se algum caso ultrapassar esse limiar, uma mensagem de erro será emitida, indicando a falha. Caso contrário, em que há uma execução bem sucedida, um certificado de resultados comparáveis é emitido, atestando a similaridade das imagens produzidas pela CPU e pela GPU.

4.4 Suporte para Diferentes Tamanhos de Grade e Blocos (CUDA)

À diferença da implementação destinada à execução no `host`, a versão para o `kernel` exige o uso de algumas subrotinas exclusivas para essa modalidade de programação. Inicialmente, reservam-se posições de memória na GPU, por intermédio da função `cudaMalloc` e da `cudaMemcpy`, alocando-se os valores de inicialização previstos em `h_resposta` para os arrays que serão manipulados na GPU. Novamente, atente-se ao fato que GPU's manipulam estrturas de dados unidimensionais e, portanto, tem-se uma transformação bijetora entre as posições de memórias matriciais suposta pela modelagem na *Iteração de Jacobi* e no array a ser utilizado na prática.

O programa, ao ser inicializado, reserva e armazena os valores do número de blocos em `b` e a quantidade de threads em cada bloco em `t`. São valores fornecidos pelo usuário.

Para dar o devido suporte para os diferentes tamanhos de grade e de blocos, posteriormente, aplica-se o tipo `dim3`, que bem especifica a dimensão de blocos e os grids de threads em um kernel. Assim, executando-se

```
dim3 threadsPerBlock(t,t)
dim3 blocksPerGrid(b,b)
```

Definimos uma representação tridimensional, mas, que possua uma grid bidimensional. Assim, conseguimos especificar quantos blocos de threads e threads por bloco serão usados em uma grade e garantimos controle sobre a forma de paralelização de tarefas entre as várias unidades de processamento.

Em particular, setam-se `t` threads e `b` blocos em cada direção X e Y dos arrays a serem manipulados e paralelizados.

A posteriori, para cada `iter`-ésima iteração, tem-se a paralelização devida do processo ao se executar a chamada à função no `kernel` para as ponteiros a posições de memória em `h_cpu`, `g_cpu`, de tamanho `n`, setando as configurações definidas pelas estrturas `dim3`.

```
jacobi_kernel<<<blocksPerGrid, threadsPerBlock>>>(h_gpu, g_gpu, n)
```

A subrotina a ser executada, `jacobi_kernel`, recebe um ponteiro `h`, um ponteiro `g` e o tamanho respectivo dessas alocações de memória.

```
__global__ void jacobi_kernel(double *h, double *g, int n)
```

Em `jacobi_kernel`, os índices de acesso e manipulação de threads e blocos são mapeados via variáveis especiais, como o `blockIdx`, `blockDim` e `threadIdx`. Estes são utilizados como combinação linear para rastrear o trabalho de cada thread e bloco para um ponto específico nos dados e para definir os índices `int i = blockIdx.y * blockDim.y + threadIdx.y + 1` e `int j = blockIdx.x * blockDim.x + threadIdx.x + 1` que serão suportes para acessar tais arrays. O processo é semelhante à versão sequencial: verifica-se se a posição é lateral e faz o cômputo do quarto da soma dos vizinhos da k-ésima posição.

Finalmente, aguarda-se a finalização e sincronização das threads, com o comando `cudaDeviceSynchronize()`, para dar prosseguimento, e encerrando-se a computação da `t`-ésima iteração, proporcionando a cópia e atualização do array `g` ao `h`, via a troca de ponteiros.

4.5 Mensuração do Tempo de Execução para Computação e Eventos da GPU

Com uma gama de ferramentas disponíveis, a biblioteca `<cuda.h>` disponibiliza funções bem úteis. Entre elas, o `cudaEventRecord`, que registra o tempo de início de um evento, e o `cudaEventSynchronize`, que aguarda o evento se completar.

Nesse viés, é possível mensurar o tempo de execução de um processo, apenas chamando a função `cudaEventElapsedTime`, atribuindo-lhe os parâmteros do tempo de início da transferência dos arquivos do `host` para o `device`, e o tempo em que esse processo se finalizou. Esse tempo médio é registrado num objeto `event_t`. Isso é visível no trecho

```
    cudaEvent_t start, stop, copy_start, copy_stop;
    cudaEventCreate(&start); cudaEventCreate(&stop); cudaEventCreate(&
        copy_start); cudaEventCreate(&copy_stop);

    cudaEventRecord(copy_start);
    cudaMemcpy(h_gpu, h_resposta, (n*n * sizeof(double *)),
        cudaMemcpyHostToDevice);
    cudaMemcpy(g_gpu, h_resposta, (n*n * sizeof(double *)),
        cudaMemcpyHostToDevice);
    cudaEventSynchronize(copy_stop);
    float time_host_to_device;
    cudaEventElapsedTime(&time_host_to_device, copy_start, copy_stop);
```

Analogamente, é possível registrar o tempo de execução do processo de transferência do `device` para o `host`.

```
    cudaEventRecord(copy_start);
    cudaMemcpy(h_resposta, h_gpu, (n*n * sizeof(double *)),
        cudaMemcpyDeviceToHost);
    cudaEventRecord(copy_stop);
    cudaEventSynchronize(copy_stop);
    float time_device_to_host;
    cudaEventElapsedTime(&time_device_to_host, copy_start, copy_stop);
```

A biblioteca também permite a mensuração do tempo de execução de algum programa que rode totalmente interna à GPU. Nesse contexto, é possível setar variáveis de eventos para inicializar e para finalizar a contagem, quando o processo é devidamente encerrado. E, ao fim, salvam-se esses valores em algum objeto `event_t`. No programa, isso pode ser constatado para cálculo do tempo de execução da *Iteração de Jacobi* com uso da GPU.

```
        cudaEventRecord(start);
        dim3 threadsPerBlock(t,t);
        dim3 blocksPerGrid(b,b);
        for (int iter = 0; iter < iter_limit; iter++) {
            jacobi_kernel<<<blocksPerGrid, threadsPerBlock>>>(h_gpu, g_gpu,
                n);
            cudaDeviceSynchronize();
            double *temp = h_gpu;
            h_gpu = g_gpu;
            g_gpu = temp;
        }
        cudaEventRecord(stop);
        cudaEventSynchronize(stop);
        float gpu_time;
        cudaEventElapsedTime(&gpu_time, start, stop);
```

Por outro lado, para a mensuração de algum processo interno à CPU, é possível o auxílio de outras bibliotecas, como a `<time.h>`. Entre essas funções estão a `clock_gettime`, que recebe uma struct `timespec` e registra o tempo atual, em segundos, em relação à Unix Epoch.

Assim, criando-se uma subrotina `calculate_elapsed_time`, que recebe duas structs do tipo `timespec`, que contém o início e o fim de um processo executado numa CPU, é possível obter o tempo de execução dessa parte do código, apenas subtraindo esses respectivos valores.

```
double calculate_elapsed_time(struct timespec start, struct timespec end)
```

4.6 Experimentos Quanto à Implementação em CUDA

A programação da *Iteração de Jacobi* foi extremamente satisfatória quanto a sua implementação em CUDA. Abaixo, serão discutidos melhor os tópicos relacionados à sua complexidade.

A fim de uma melhor compreensão dos experimentos, defina como valores padrão Número de Threads (t) = 16, Número de blocos (b) = 16, Número de Pontos (n) = 100 e Número de iterações (iter) = 100. Adicionalmente, para cada situação, foram executadas 10 vezes, para se obter um intervalo estatístico de confiança.

4.6.1 Tempo de Execução Em Função de N

Foram coletados alguns dados quanto ao tempo de execução na GPU e na CPU do programa `heat-tarefa2.cu`, modificando os valores de N para $\{50, 75, 100, 125, 150, 175, 200, 225, 250\}$. Para os valores médios, é possível setar o seguinte gráfico.

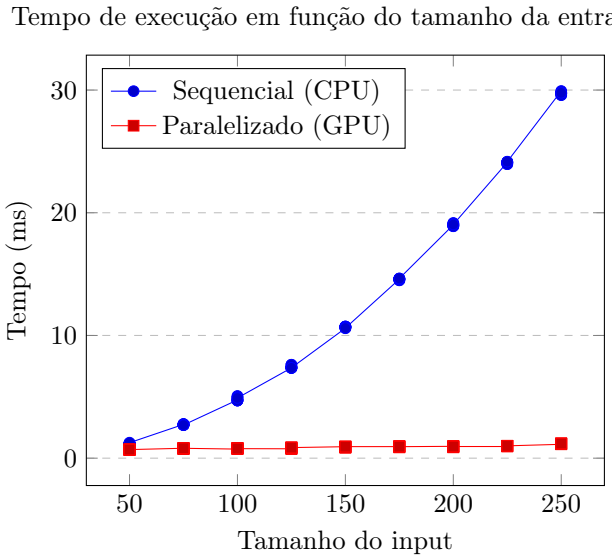


Figure 1: Tempo de execução em função de N

Como é possível notar, quanto maior o tamanho dos inputs, isto é, os arrays a serem modelados, maior a discrepância do tempo de execução entre a CPU e a GPU. A versão sequencial tende a uma classe de complexidade maior do que a paralelizada, em outras palavras, a primeira tende a ser linear $O(N)$, enquanto a outra, constante $O(1)$.

4.6.2 Host-Device e Device-Host Em Função de N

Adicionalmente, é possível observar o tempo médio de transferência de dados entre o `device` e o `host`.

A priori, conforme segue do gráfico obtido dos dados coletados, é possível observar que, em toda ocasião, o tempo de transferência do `Host` para o `Device` é maior que o processo contrário. E, isso não é por acaso. Note que no programa `heat-tarefa2.cu`, fazemos duas transferências do tipo `HD` (do `host` para `h` e para `g`), e apenas uma do tipo `DH` (do `h` para `host`).

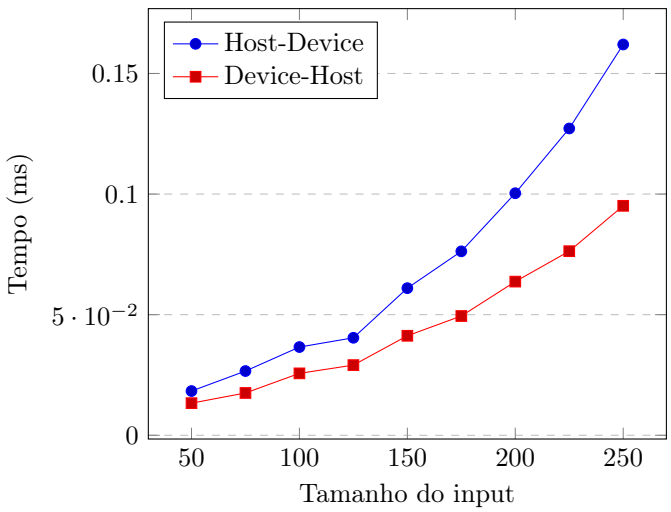


Figure 2: Tempo de transferência

Abaixo, se seguem as tabelas das estatísticas médias das transferências dos tipos `HD` e `DH`, acrescidas dos intervalos de confiança para o tempo de execução respectivos.

N	median	mean	std	min	max	Q0	Q1	Q3	Q4
50	0.018944	0.019251	0.002254	0.016384	0.022528	0.016384	0.017408	0.020992	0.022528
75	0.052224	0.048538	0.012394	0.029696	0.071680	0.029696	0.041216	0.054016	0.071680
100	0.060720	0.057130	0.010236	0.037856	0.063872	0.037856	0.060368	0.062880	0.063872
125	0.078880	0.074880	0.009942	0.059840	0.085984	0.059840	0.066944	0.080176	0.085984
150	0.097968	0.091034	0.016766	0.060896	0.106176	0.060896	0.078616	0.102664	0.106176
175	0.125264	0.123718	0.013341	0.087104	0.133280	0.087104	0.124520	0.130760	0.133280
200	0.159216	0.160144	0.003894	0.155648	0.166272	0.155648	0.156920	0.163088	0.166272
225	0.193408	0.188934	0.016806	0.141248	0.196480	0.141248	0.193136	0.194928	0.196480
250	0.239344	0.230083	0.023563	0.169280	0.243360	0.169280	0.237456	0.241608	0.243360

Table 2: Tabela contendo as estatísticas da transferência de HD, em milissegundos

N	Média	IC
50	0.019251	[0.02, 0.02]
75	0.048538	[0.04, 0.06]
100	0.057130	[0.05, 0.06]
125	0.074880	[0.07, 0.08]
150	0.091034	[0.08, 0.10]
175	0.123718	[0.12, 0.13]
200	0.160144	[0.16, 0.16]
225	0.188934	[0.18, 0.20]
250	0.230083	[0.22, 0.24]

Table 3: Tabela contendo os IC da transferência de HD, em milissegundos

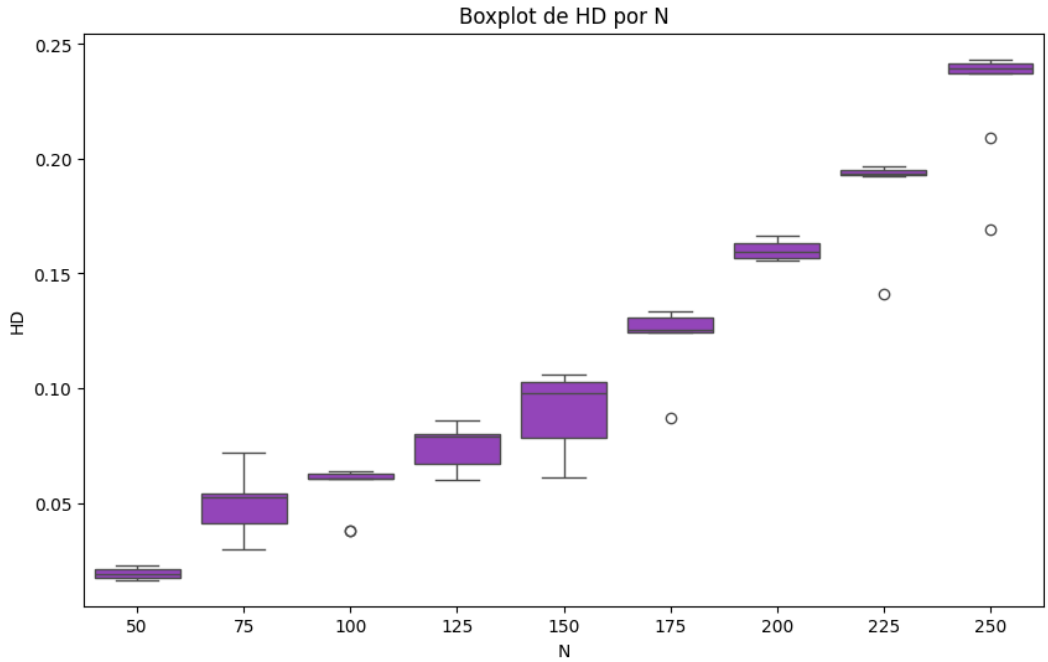
N	median	mean	std	min	max	Q0	Q1	Q3	Q4
50	0.015968	0.016090	0.000524	0.015392	0.017248	0.015392	0.015824	0.016144	0.017248
75	0.025072	0.025309	0.005664	0.019808	0.038400	0.019808	0.020760	0.026760	0.038400
100	0.039536	0.036253	0.006377	0.026784	0.041312	0.026784	0.030256	0.040584	0.041312
125	0.044336	0.044672	0.004512	0.040096	0.049568	0.040096	0.040512	0.048904	0.049568
150	0.060336	0.055542	0.006809	0.047136	0.062304	0.047136	0.047888	0.060536	0.062304
175	0.076448	0.074947	0.005903	0.058560	0.078752	0.058560	0.075712	0.077496	0.078752
200	0.092672	0.090832	0.007212	0.070656	0.095424	0.070656	0.091784	0.093344	0.095424
225	0.111968	0.112566	0.002369	0.108448	0.116672	0.108448	0.111264	0.114120	0.116672
250	0.133920	0.131427	0.010510	0.102080	0.138208	0.102080	0.132760	0.135568	0.138208

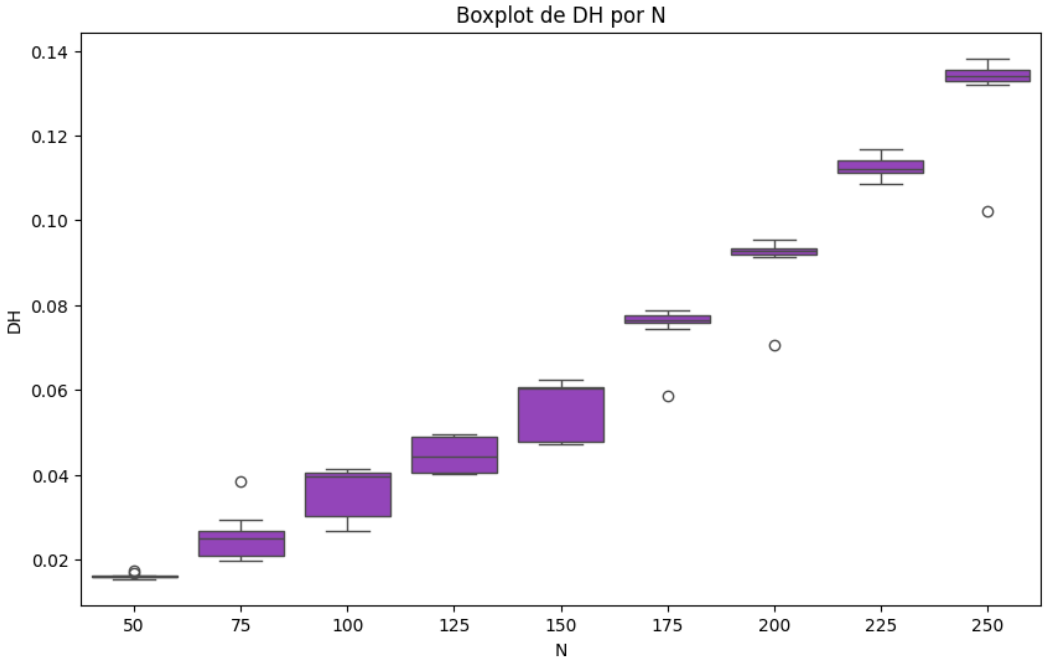
Table 4: Tabela contendo as estatísticas da transferência de DH, em milissegundos

N	Média	IC
50	0.019251	[0.02, 0.02]
75	0.048538	[0.04, 0.06]
100	0.057130	[0.05, 0.06]
125	0.074880	[0.07, 0.08]
150	0.091034	[0.08, 0.10]
175	0.123718	[0.12, 0.13]
200	0.160144	[0.16, 0.16]
225	0.188934	[0.18, 0.20]
250	0.230083	[0.22, 0.24]

Table 5: Tabela contendo os IC da transferência de DH, em milissegundos

Os resultados se mostraram bem consistentes, com um desvio padrão baixo relativo, mesmo para dimensões maiores da instância do problema.





Apesar de alguns outliers como previstos na plotagem desses BoxPlots, os tempos de execução de mostraram dentro do esperado. Alguns com mais variância que os outros (como para N = 150), mas, no geral, os valores foram próximos entre si, demonstrando uma tendência de crescimento ao longo das dimensões.

É importante notar que essa seção do programa é sequencial, a fim de não alterar os dados a serem manipulados. Assim, torna-se fundamental levar essa parte em consideração, pois é uma constante que não pode ser modificada na paralelização do programa.

4.6.3 Tempo de Execução Em Função de b

Moderação, talvez essa seja a mehor palavra para descrever a quantidade estipulada no uso de blocos no grid. A seguir, verifique o impacto do valor de `b` ao longo dos experimentos. Nesta bateria de experimentos, variou-se o número de blocos do grid.

Abaixo, os experimentos da blocagem se deram numa faixa bem baixa de valores, entre 10 a 30. Note que, consideravelmente, o programa paralelizado teve um desempenho muito superior ao sequencial, com uma vantagem superior a 3ms de execução.

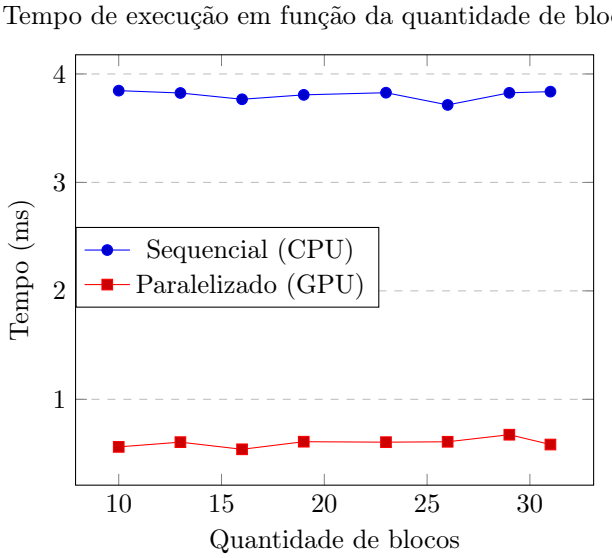


Figure 3: Tempo de execução em função de b

Contudo, para valores maiores, que tendessem a 5000, essa situação se reverteu, sendo consideravelmente melhor a execução sequencial à paralela. Isso se deve à ociosidade e alta concorrência entre os processos, mais atrapalhando do que favorecendo um bom desempenho.

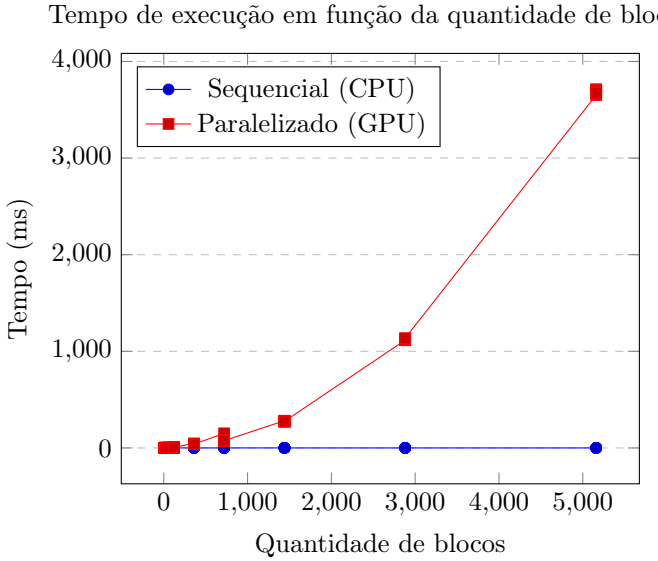


Figure 4: Tempo de execução em função de b

A conclusão é que, moderada e adequadamente à modelagem do problema, é sensato o aumento da quantidade dos blocos. Contudo, essa taxa não é linear para se paralelizar de forma satisfatória um programa, possibilitando programas paralelizados corretos, mas, com a complexidade comprometida.

4.6.4 Tempo de Execução Em Função de t

O próximo experimento exigiu a variação da quantidade de Threads por blocos, os quais variaram entre $\{10, 13, 16, 19, 23, 26, 29, 32\}$.

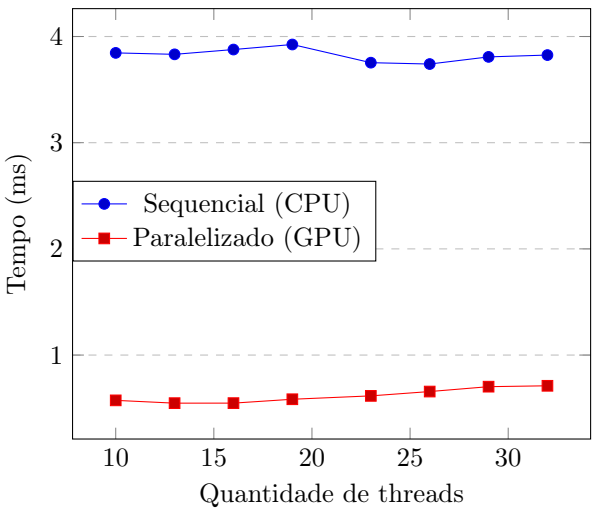


Figure 5: Tempo de execução em função de t

Para essa faixa de valores, dados os valores definidos por padrão, aumentar a quantidade de Threads se mostrou um fator satisfatório, sobretudo, porque o tempo de execução da versão sequencial se manteve, cerca, de 3ms maior do que a paralelizada.

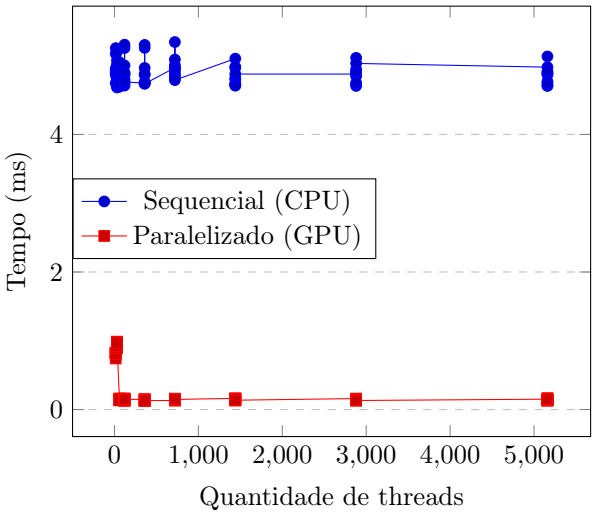


Figure 6: Tempo de execução em função de t

Contudo, para mais Threads, o padrão se manteve. Assim, o ambiente do `kernel` suporta muitas Threads, o que não derruba o desempenho do programa. Todavia, por se manter estável, não é muito lógico, eficiente e ecológico manter tantas Threads, quando se não há uma melhora significativa que motivem um porque de estarem lá.

4.6.5 SpeedUP em Relação a N

O quão mais potente um programa rodado numa GPU pode ser em relação à CPU? Sob essa dúvida muito intrigante, é possível ter a resposta com a fórmula derivada da lei de Amdahl, $SpeedUP(N) = \dfrac{1}{(1-P) - \dfrac{P}{N}}$, onde P corresponde à parte paralelizável, e N, o número de workers.

Contudo, a termos práticos, como foi possível paralelizar todo o processo, então, a fórmula a ser utilizada será $SpeedUP(N) = \frac{Tempo_{CPU}}{Tempo_{GPU}}$.

Abaixo, se segue a curva obtida pelo cálculo do SpeedUP.

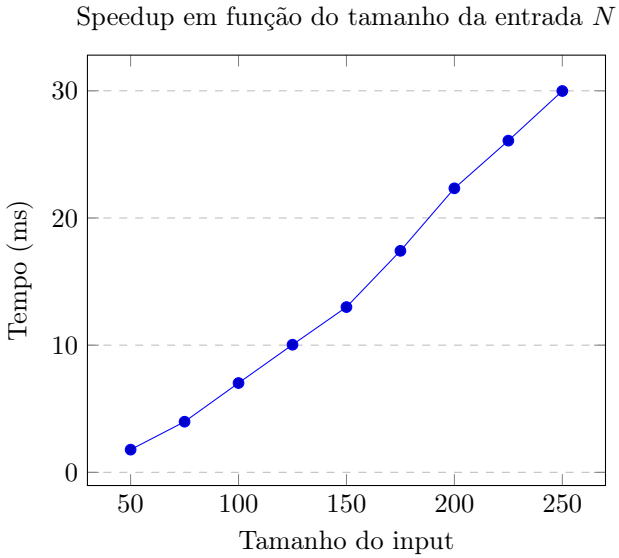


Figure 7: Speedup do programa da tarefa 2

E as estatísticas médias, acompanhadas do IC:

N	median	mean	std	min	max	Q0	Q1	Q3	Q4
50	1.705018	1.716595	0.039457	1.667871	1.804888	1.667871	1.698464	1.715035	1.804888
75	3.411724	3.453610	0.114662	3.334702	3.711596	3.334702	3.379687	3.507553	3.711596
100	6.098862	6.088659	0.255491	5.763733	6.457018	5.763733	5.888469	6.284559	6.457018
125	8.776930	8.985142	0.468548	8.548048	9.896640	8.548048	8.691744	9.229882	9.896640
150	11.598728	11.704776	0.422516	11.275602	12.626267	11.275602	11.411551	11.888443	12.626267
175	15.467132	15.512367	0.433042	15.025696	16.620170	15.025696	15.276870	15.589781	16.620170
200	20.062985	20.156835	0.571353	19.717643	21.662629	19.717643	19.786581	20.243271	21.662629
225	24.000776	24.120692	0.472208	23.750970	25.407235	23.750970	23.903912	24.139441	25.407235
250	25.193946	25.286768	0.442501	24.961755	26.488027	24.961755	25.056246	25.270053	26.488027

Table 6: Tabela contendo as estatísticas do SpeedUP(N), em milissegundos

N	Média	IC
50	1.716595	[1.69, 1.74]
75	3.453610	[3.38, 3.52]
100	6.088659	[5.93, 6.25]
125	8.985142	[8.69, 9.28]
150	11.704776	[11.44, 11.97]
175	15.512367	[15.24, 15.78]
200	20.156835	[19.80, 20.51]
225	24.120692	[23.83, 24.41]
250	25.286768	[25.01, 25.56]

Table 7: Tabela contendo os IC do SpeedUP(N), em milissegundos

Como é possível notar, para entradas cada vez maiores, o speedUP se torna mais expressivo, demonstrando que a CPU pode obter desempenhos cada vez piores, ao passo que a GPU se demonstre mais constante e melhores. A curva segue uma proporção próxima à obtida somente pela CPU, evidenciando o poder computacional de uma

GPU ao se manter constante.

5 Tarefa 3

5.1 Implementação da Distribuição de Calor com Corpo Quente

O código, essencialmente, é o mesmo que o da versão anterior. Mas aqui, estipulamos, inicialmente, as dimensões desse corpo quente no quarto, definindo-os no início do programa

```
  BODY_TEMP 37
BODY_START_X x_i
BODY_START_Y y_i
BODY_END_X x_f
BODY_END_Y y_f
```

onde x_i, x_f, y_i, y_f definem os parâmetros de modo hardcode.

Assim, definidas essas dimensões, a diferença para a versão anterior é que se inicializa a região do corpo com a temperatura correspondente a 37^o e, nas, futuras *Iterações de Jacobi* (tanto da versão paralelizada quanto da sequencial), preserva-se e protege-se a modificação à essa região.

5.2 Experimentos Entre Versões

Defina como a temperatura padrão, 37^o, e um corpo retangular de dimensões, $x_i = 14, x_f = 19, y_i = 30, y_f = 35$. Portanto, com tamanho 6x6 e área proporcional à 36.

Executando, para os mesmos parâmetros definidos em `b` $= 16$, `t` $= 16$, `iter` $= 100$, mas, com variância sobre `N`, obtém-se os seguintes gráficos comparativos:

Em relação ao cálculo sequencial, a tarefa 3 demandou um maior custo de recursos computacionais. Isso porque, apesar de não preencher o interno da figura e executar a computação do quarto da soma dos vizinhos, o código em `heat-tarefa3.cu` faz $\theta(N^2)$ comparações quanto à verificabilidade de que determinada região pertence ao corpo. Contudo, não foi um desempenho tão grotesco e relativamente satisfatório.

Tempo de execução (sequencial) em função do tamanho da entrada N

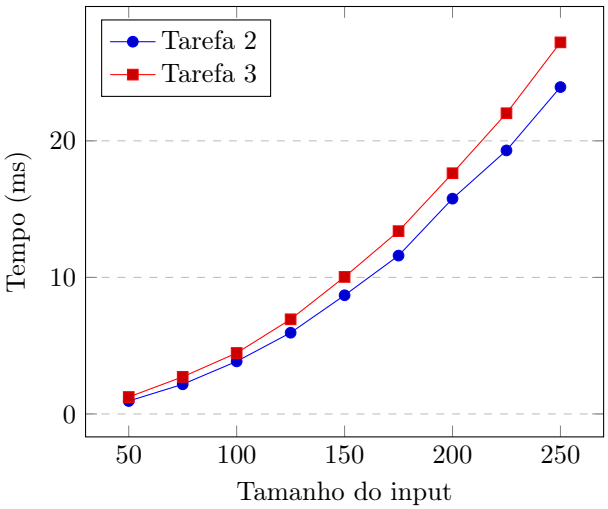


Figure 8: Tempo de execução (sequencial) em função de N

Notadamente, a tarefa 3 foi comparável à 2, no quesito da GPU, uma vez que apenas se restringem as partes iteráveis. Assim, os desempenhos foram bem próximos entre si.

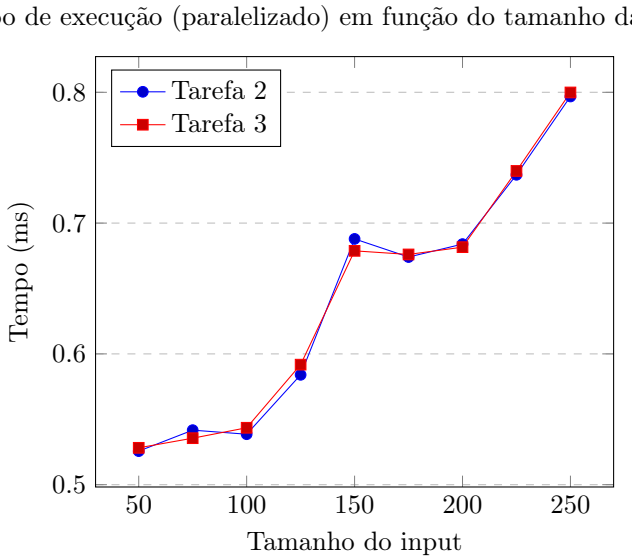


Figure 9: Tempo de execução (paralelizado) em função de N

Assim, obtém-se o gráfico comparatíveis de speedUP, entre as versões sequenciais e paralelizadas da Terefa 3 e da Tarefa 2.

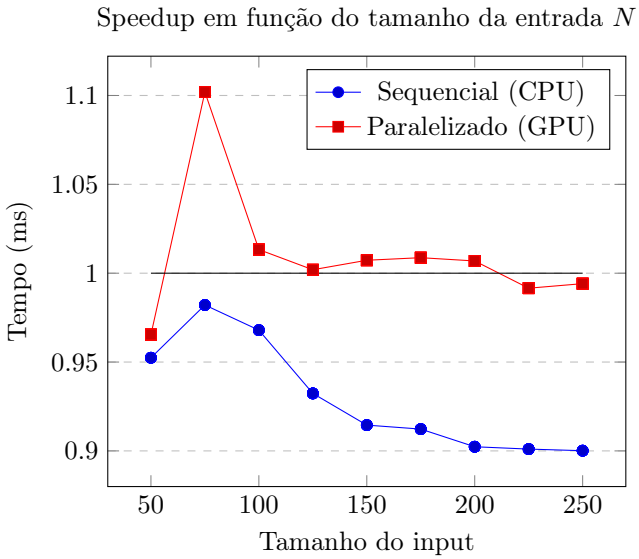


Figure 10: Speedup da tarefa 3 em relação à tarefa 2

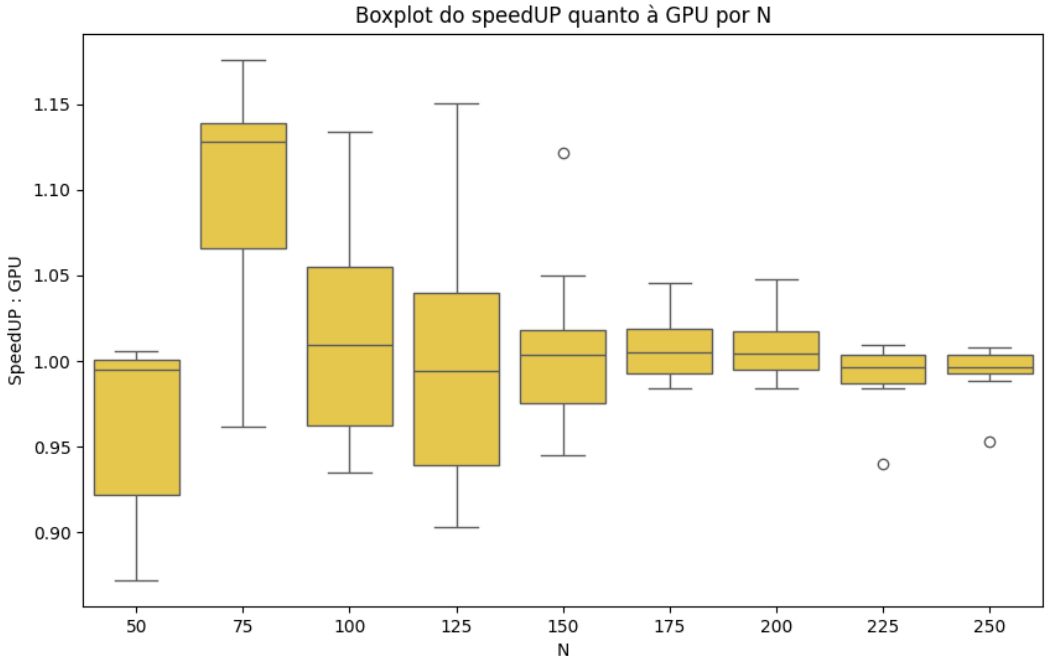


Figure 11: Speedup da tarefa 3 em relação à tarefa 2, quanto à GPU

Note que o speedUP da GPU oscila próximo de 1. Isso indica que a paralelização entre a tarefa 2 e a tarefa 3 não está gerando ganhos significativos de desempenho, explicado muito provavelmente pelo fato da única modificação significativa na tarefa 3 ser a comparação extra no `kernel`, para que a iteração não modifique a temperatura do corpo quente. Essa comparação extra, num contexto paralelizado, não parece alterar muito o desempenho do programa.

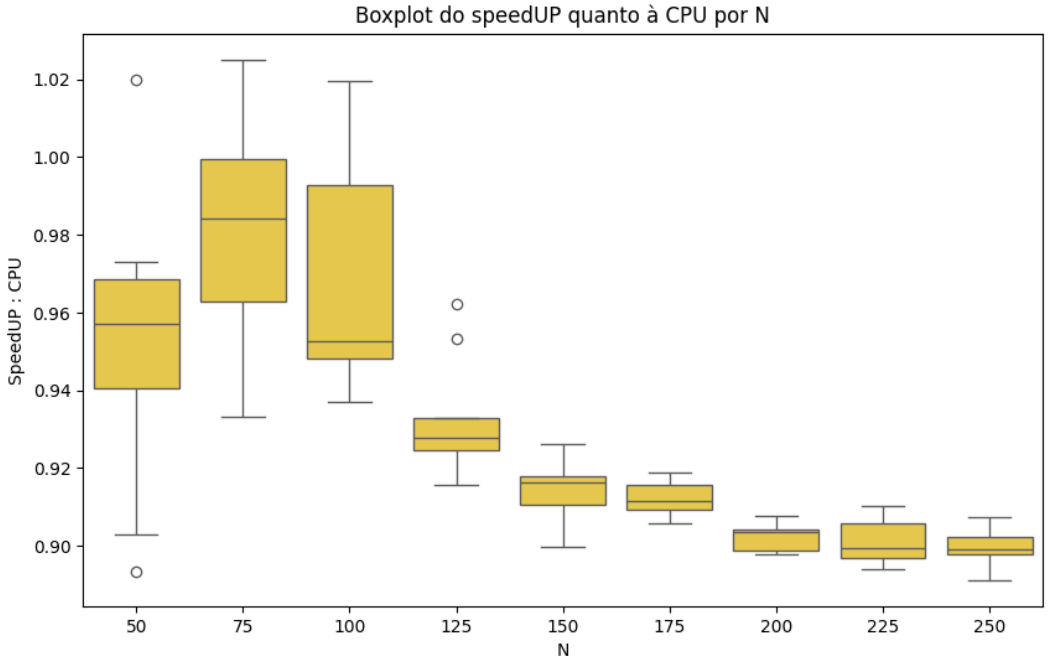


Figure 12: Speedup da tarefa 3 em relação à tarefa 2, quanto à GPU

É interessante reparar como o speedUP da tarefa 3 em relação a tarefa 2 é, geralmente, menor que 1, o que indica que a tarefa 3 possui uma tendência a ser mais lenta que a tarefa 2 num contexto sequencial. Isso muito provavelmente se explica pela comparação extra a ser realizada, porque ela é feita $N^2 \times$ `iter` vezes.

Finalmente, ilustra-se o speedUP interno entre GPU e CPU em relação somente à tarefa 3. Note que é um

gráfico muito semelhante ao da tarefa 2.

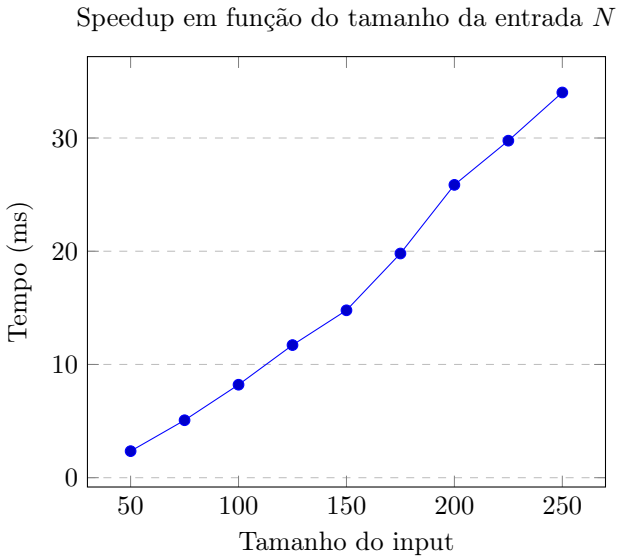


Figure 13: Speedup do programa da tarefa 3

5.3 Desempenho Quanto ao Tamanho dos Corpos

Também é possível analisar o desempenho alterando o tamanho do corpo quente. Nesse viés, investigando-se para os padrões de `6x6` e `21x21` (a diagonal principal indo de (14,30) até (34,50)), tem-se os seguintes comparativos

A versão sequencial do programa se mantém para tamanhos expressivamente distintos. Note que apesar do tamanho do corpo ter crescido em 12.25 (saiu de 36 pixeis para 441), o tempo de execução não teve alterações expressivas.

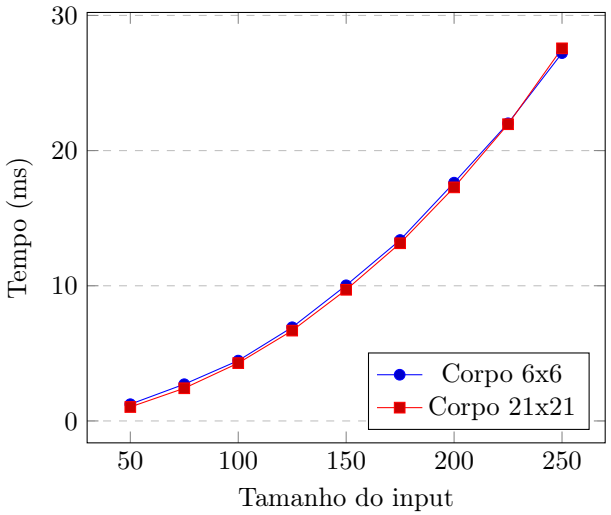


Figure 14: Tempo de execução (sequencial) em função de N

Ademais, na versão paralelizada, torna-se um processamento tão rápido, que o tamanho do corpo se tornou irrelevante.

Comparação dos tempos de execução (paralelizado) dos corpos maior e menor em função do tamanho da entrada N

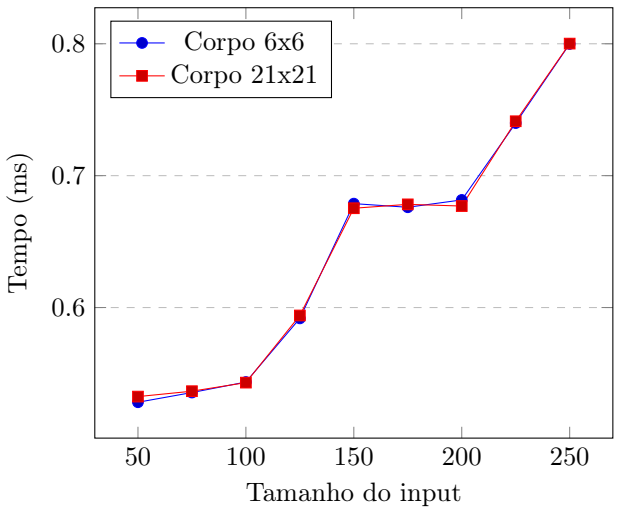
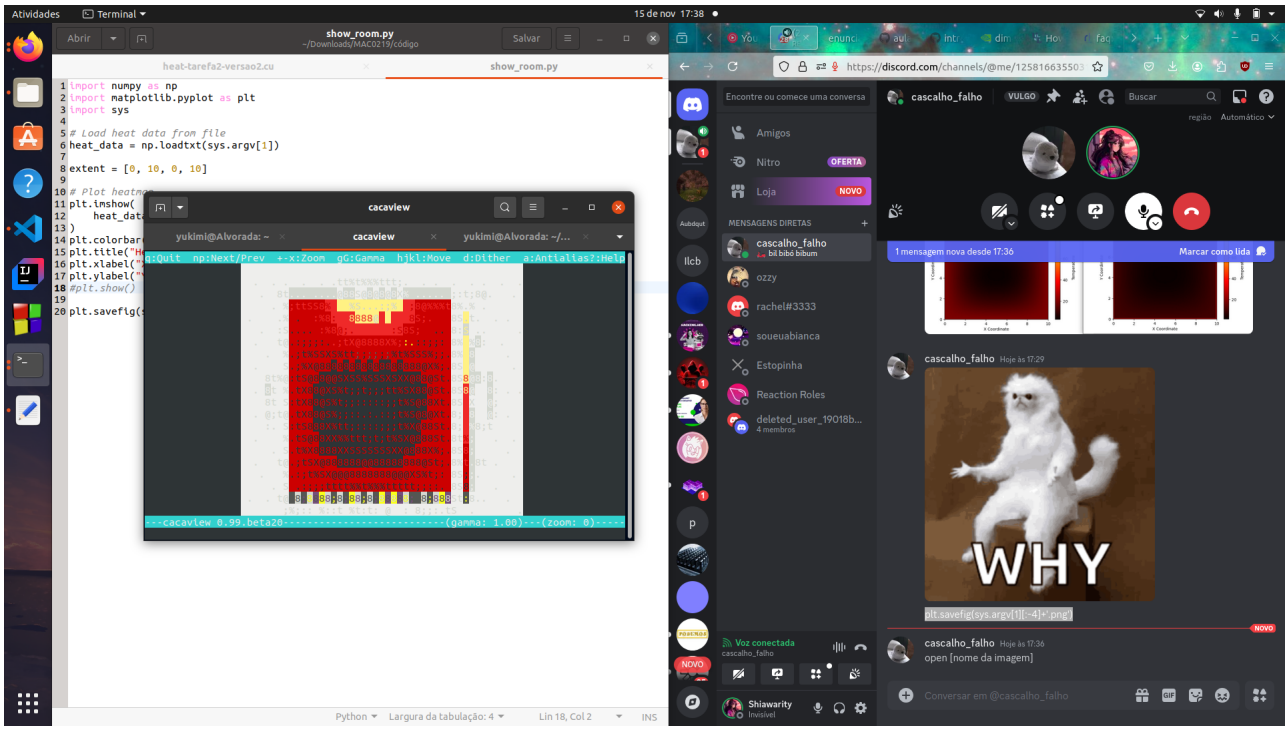


Figure 15: Tempo de execução (paralelizado) em função de N

6 Desafios encontrados

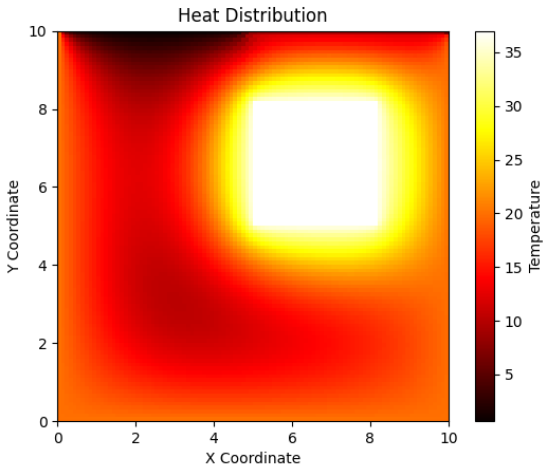
Em primeiro lugar, veio a dificuldade de entender qual parte era melhor para paralelizar. A princípio, pensamos em pararelizar as iterações, mas após perceber que isso não produziria resultados corretos, resolvemos paralelizar nas regiões da imagem.

Um dos desafios mais frustrantes foi a visualização das imagens geradas. Como estávamos fazendo o EP via SSH na Rede Linux, toda vez que uma imagem era gerada nós tínhamos que exportá-la de volta para nossa máquina, porque a Rede Linux não possui instalado no sistema qualquer comando para visualização de imagens senão o `open`, que, pela screeshot abaixo (feita acidentalmente, mas comicamente), não fazia uma boa visualização:



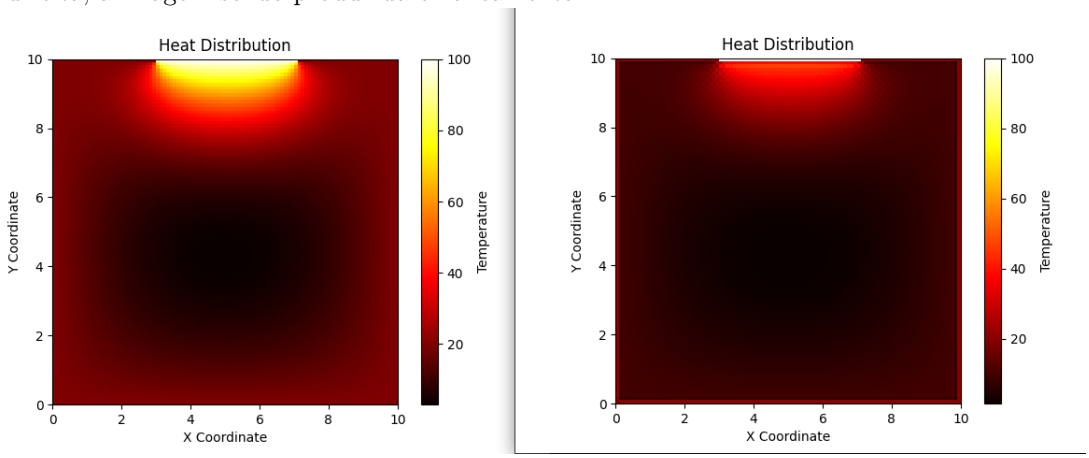
Eventualmente fizemos um script para que a cada execução do código do EP, a imagem gerada era colocada no nosso site privado da `linux.ime.usp.br`. Isso salvou várias dores de cabeça.

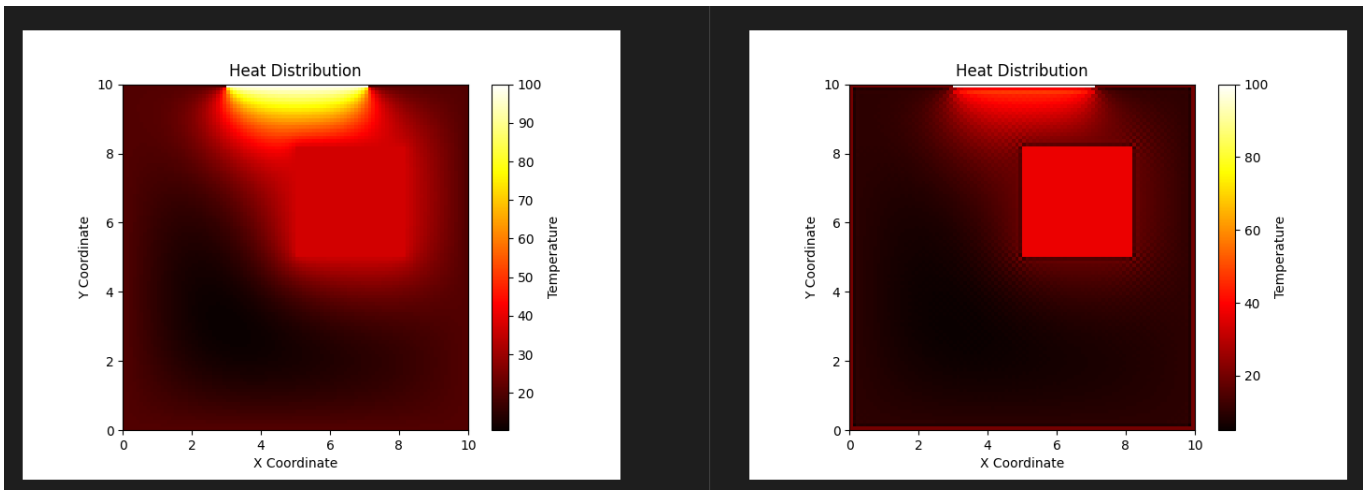
Uma outra dificuldade encontrada foi, em partes, entender o enunciado da tarefa 3. Não sabíamos se precisávamos manter a lareira ou apenas o corpo quente. Essa é uma imagem de quando tentávamos fazer sem a lareira:



Eventualmente conseguimos tirar essa dúvida com um monitor, que disse que era para manter a lareira.

O pior e mais frustrante desafio foi um bug no código da tarefa 2 (e consequentemente na tarefa 3), na versão paralelizada. Ele envolvia as iterações não se sobreporem corretamente. Veja as imagens. À esquerda, o esperado; à direita, a imagem sendo produzida erroneamente.





Esse erro só aparecia na versão paralelizada. E por quê? Passando linha por linha no código, refatorando, refazendo tudo, sempre acontecia esse mesmo erro. Depois de alguns bons *dias* pensando, e com esclarecimentos de um monitor, conseguimos encontrá-lo: O problema era nesse trecho do código:

```
1     dim3 threadsPerBlock(t,t);
2     dim3 blocksPerGrid(b,b);
3     for (int iter = 0; iter < iter_limit; iter++) {
```

```
        jacobi_kernel<<<blocksPerGrid, threadsPerBlock>>>(h_gpu, g_gpu, n);
        cudaDeviceSynchronize();

        double *temp = h_gpu;
        h_gpu = g_gpu;
        g_gpu = temp;
    }
```

O problema no código foi que esquecemos de inicializar o `g_gpu` antes de começar as iterações. No kernel, usamos dois vetores, `h_gpu` e `g_gpu`, onde `h_gpu` contém os valores antigos para leitura e `g_gpu` recebe os novos valores para escrita a cada iteração. Geralmente não haveria problema em deixar `g_gpu` sem inicialização, porque é apenas para escrita. Entretanto, como o kernel escreve apenas "no miolo" (não nas bordas) da imagem, então, no final de cada iteração, como trocamos os ponteiros para que na próxima iteração os novos valores sejam calculados com base nos antigos, então o novo vetor de leitura `h_gpu` (antigo `g_gpu`) agora tem as bordas da imagem indefinidas. Então os cálculos das iterações seguintes eram feitos de forma errada.

Nós concertamos isso ao fazer duas transferências de vetor do host pro device; uma para `h_gpu` e ou para `g_gpu`.

Por fim, no fim, ficamos com dúvida sobre como calcular o speedup. Se era apenas para fazer uma razão simples entre o tempo de execução da versão paralela e o da versão sequencial ou se precisávamos fazer as contas propostas pela lei de Amdahl. Essa dúvida foi rapidamente resolvida por um monitor, que disse que era a primeira opção.

7 Comparações Entre GPU's

Finalmente, é possível comparar o desempenho entre diferentes GPU's. Essencialmente, a máquina em que todos os experimentos foram testados derivou da `linux.ime.usp.br`. Esta contando com as seguintes especificações

CPU	AMD Ryzen 7 5700G with Radeon Graphics
RAM	125Gi
SO	Debian GNU/Linux 12 (bookworm)
Placa de Vídeo	NVIDIA GeForce RTX 3060

Table 8: Especificações da Rede Linux

Será testada uma segunda máquina, do `Google Colab`, com as seguintes especificações

CPU	Intel(R) Xeon(R) CPU @ 2.20GHz
RAM	12Gi
SO	22.04.3 LTS (Jammy Jellyfish)
Placa de Vídeo	Tesla T4

Table 9: Especificações do Google

O experimento será dado em relação ao tamanho da entrada dos arrays a serem processados, e, portanto, `b = 16, t = 16, iter= 100`, para ambas situações, referentes à tarefa 2.

Comparação dos tempos de execução (paralelizado) dos corpos maior e menor em função do tamanho da entrada N

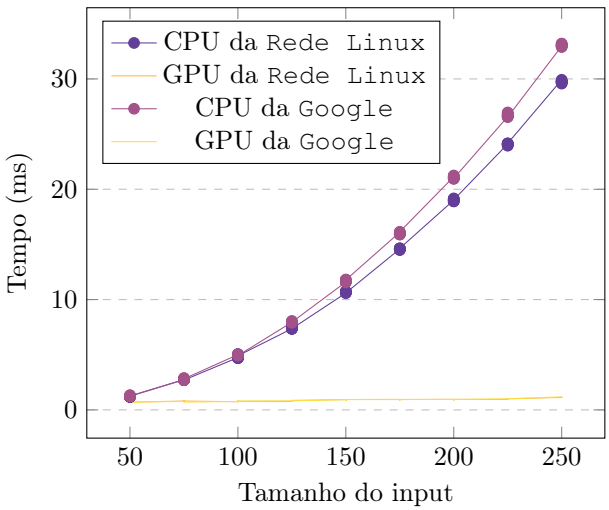


Figure 16: Tempo de execução (paralelizado) em função de N

É possível observar que, quanto à CPU, o modelo disponibilizado pela `Rede Linux` foi melhor do que a da `Google`. Contudo, quanto às GPU's, ambas foram extremamente eficientes.

8 Conclusão

Após essas implementações e experimentos, foi possível observar e constatar, portanto, a eficiência no uso de GPU's para paralelização de tarefas.

O uso de GPU's na implementação dos programas demandou algumas ferramentas disponíveis na biblioteca `CUDA-Toolkit`, como o `cudaMalloc` e o `cudaMemcpy`, para reservar e manipular um espaço de memória da GPU, `cudaEventCreate`, `cudaEventRecord`, `cudaEventSynchronize`, `cudaEventElapsedTime` para cálculo devido do tempo de execução do programa, e a estrutura `dim3` para definir os parâmetros quanto à blocagem e o número de threads para execução no `kernel`.

Alguns experimentos feitos indicaram o impacto sobre o tamanho da entrada dos inputs e o tempo de execução de um programa sequencial (na CPU) e de um paralelizado (na GPU). Definitivamente, a paralelização do código tornou o processo extremamente eficiente, a ponto de tornar sua complexidade constante em relação ao primeiro (mais linear).

Ademais, um tempo considerável do processo é a transposição de dados entre `host` e `device`, que deve ser levado em consideração na arquitetura da sistema. Note que esse valor é constante e não é passível de paralelização, para acelerar isso.

Também é interessante perceber que alterar de modo extremal a quantidade de `blocos` pode piorar a complexidade do programa, em questão de concorrência, mas, que, numa taxa moderada é benéfico e extremamente eficiente ao programa. Por outro lado, é igualmente viável ter controle sobre a quantidade, manipulação e injeção de Threads. Embora não possam ser prejudiciais à paralelização, muitas delas não bonificam tanto a complexidade final e podem ter um custo considerável computacional desperdiçado.

Assim, para a primeira tarefa, houve um speedUP significativamente positivo entre o uso da CPU e da GPU.

A seguir, com fixação de uma região de temperatura constante, foi possível verificar que houve uma leve piora na complexidade do programa, em relação à versão sem um corpo quente, executada na CPU. Contudo, sua versão paralelizada teve um desempenho igualmente satisfatório.

Alguns desafios foram encontrados, como a manipulação de dados e imagens via `SSH`, especificações do enunciado e bugs. Em especial, uma relacionada à transferência de dados entre o `host` e o `device`. E a arquitetura

da fórmula esperada para o speedUP.

Finalmente, encerra-se esse relatório com uma comparação entre máquinas. A parte comparável se resume à CPU, uma vez que, na GPU, ambas as máquinas testadas tiveram desempenho igualemente satisfatório.