Implementação Paralela de Convolução em Imagens
Relatório Final
Programação Concorrente - 2024/1

Matheus da Cruz Percine Pinto [1], **Manoel Marcelo da Silva**[1]

[1] Instituto de Computação – Universidade Federal do Rio de Janeiro
Rio de Janeiro, Brasil.

{matheuscpp,manoelms}@ic.ufrj.br

1. Descrição do problema geral

Na computação de alto desempenho, o processo de convolução é uma operação de matriz bastante popular usado principalmente no processamento de sinais, na computação visual e no processamento de imagens e de vídeos. Nessas áreas, ele é responsável por aplicar filtros em valores mais desejados para determinada aplicação, como por exemplo, é usado na Rede Neural Convolucional (CNN) na detecção de padrões, principalmente em imagens.[1]

No entanto, tal processo convolucional é bastante custoso computacionalmente, já que a convolução tipicamente envolve bastante cálculos para cada elemento de uma matriz. Imagine o processamento de uma imagem comum com tamanho 4000x4000, tal matriz possui quase 16 milhões de valores para cada canal de cor. É possível perceber o tempo que seria gasto de forma sequencial, assim, para a viabilidade do processamento de grandes imagens e de vários algoritmos, como o CNN e o processamento em tempo real, a implementação de forma concorrente se torna essencial.

1.1. Convolução em uma imagem

O processo de convolução se baseia na modificação de um número específico com base em seus vizinhos mais próximos. No contexto de uma imagem em escala de cinza, a imagem é representada por uma matriz bidimensional onde cada valor corresponde à intensidade de um pixel. Caso seja colorida, temos outras duas matrizes - três no total - para os canais de cores primárias vermelho, verde e azul. Diante disso, para fazer uma convolução deve-se calcular para cada pixel(valor) para cada canal, em que cada pixel assume um valor no intervalo de 0-255.

A fórmula convolucional para duas dimensões é dada por

$$g(x,y) = \omega * f(x,y) = \sum_{i=-a}^{a} \sum_{j=-b}^{b} \omega(i,j) f(x-i, y-j)$$

$g(x,y)$ é a imagem filtrada, $f(x,y)$ é a imagem original, e ω é o kernel do filtro. Cada elemento do kernel do filtro é considerado por $-a \leq i \leq a$ e $-b \leq j \leq b$.

A convolução envolve a aplicação de uma matriz menor, chamada de kernel, sobre a imagem. Cada kernel possui propriedades úteis, como borrar uma imagem ou reforçar bordas horizontais. O kernel possui ”pesos” que determinam a contribuição de cada pixel

vizinho na geração do novo valor para o pixel central. Um exemplo de kernel é o Gaussiano, que calcula o valores dos pixels usando uma distribuição gaussiana em relação aos pixels vizinhos - por isso seu nome - que promove uma suavização ou borra a imagem final. Nesse caso, quanto maior o kernel, ou seja suas dimensões, maior influência os pixels vizinhos farão na imagem final (considerando um desvio padrão alto o suficiente), logo mais borrado ficará no resultado final.

Figure 1. Kernel gaussiano aplicado em uma imagem.

Durante o processo, o kernel é posicionado sobre a imagem original, com o centro do kernel alinhado a um pixel específico da imagem e, em seguida, para cada posição do kernel, calcula-se o produto entre cada elemento do kernel e sua posição correspondente da imagem, somando todos esses produtos para obter o novo valor do pixel. Finalmente, o kernel é então deslocado para a próxima posição (geralmente de forma sequencial, pixel a pixel) até que toda a imagem tenha sido processada. Este processo é repetido para cada canal de cor, no caso de imagens coloridas.

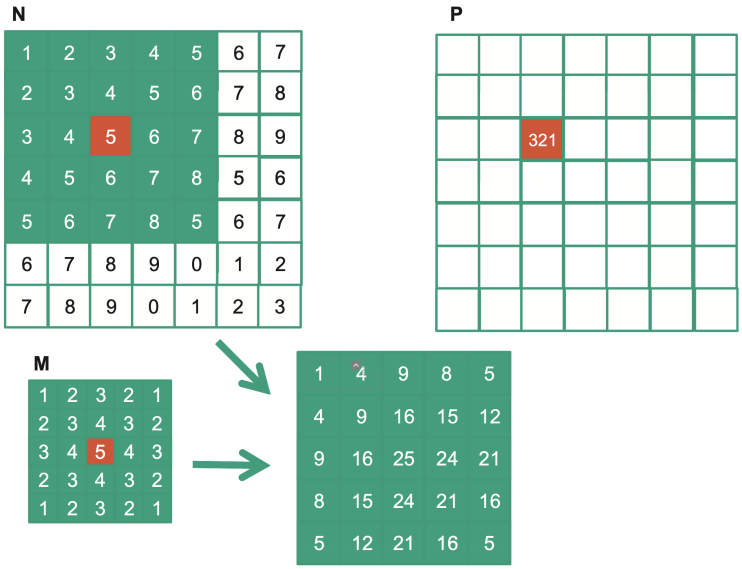

Figure 2. Exemplo de Convolução em uma matriz em $\mathbb{R}_2$.

Um dos possíveis problemas nesse processo que pode acontecer é no caso de um dos valores acessados ficarem fora do limite da matriz original. Isso acontece no momento em que o pixel a ser calculado fica em uma das bordas da imagem, assim, parte dos valores

vizinhos que o kernel tenta calcular está fora da imagem. Para resolver esse impasse, é feito o processo chamado de padding, que consiste em aumentar as bordas da imagem com zero até que não seja possível ocorrer o problema descrito anteriormente. Outra forma de fazer esse processo é fazer uma checagem a cada leitura da matriz original para evitar acessos indevidos à memória, economizando memória já que não aumenta a imagem desnecessariamente. Por este motivo, esse ultimo método é o que será implementado no projeto.

Portanto, um algoritmo de convolução deve receber como parâmetros uma matriz de entrada com os dados de uma matriz N a ser convolucionada, nesse caso os dados de uma imagem, uma matriz outra de kernel M e a matriz em que será escrita P.

2. Projeto e implementação da solução concorrente

Como vimos, a convolução é simplesmente uma operação em que é aplicada para cada elemento de uma matriz a soma ponderada de seus elementos vizinhos com seus pesos definidos pelo kernel. De acordo com a definição de Navarro et al[2], como podemos aplicar a convolução no conjunto de elementos de uma imagem para formar a convolução na imagem inteira, ou seja, como a matriz N é composto de elementos de dados e é possível aplicar a convolução para todo o domínio, neste problema claramente ocorre um paralelismo de dados.

Além disso, há somente a leitura nas matrizes de entrada, logo não há nenhum problema de diversos fluxos de execução executarem em paralelo a convolução. Ademais, também não há nenhuma dependência de dados entre o cálculo de cada elemento da matriz entre si, nem a atualização de nenhum elemento anterior, já que a matriz resultante não sobrescreve a original e o processamento não usa os dados novos.

Portanto, é visível o potencial uso de uma implementação paralela por conta do paralelismo de dados existente.[1]

Para isso, deve-se haver uma preocupação de garantir que todas as unidades de processamento recebam a mesma carga de trabalho para aproveitar ao máximo todos os recursos de processamento disponíveis. Logo, todas as estratégias de concorrência garantirão tal princípio.

A implementação será feita primariamente em Python e também em C para motivos de comparação de eficiência e performance entre as linguagens.

2.1. Python

O Python possui uma performance em threads bastante inferior em relação à linguagem C, isso muito de deve por conta do Global Interpreter Lock (GIL) que impede que quaisquer duas threads executem ao mesmo tempo (usem a CPU). Nesse contexto, é ideal usar concorrência com processos por ser muito CPU-Bound, ou seja, o tempo de processamento depende principalmente do processador e não de I/O.

Porém, usando processos não é possível gravar diretamente na matriz de saída por conta das particularidades de processo. Há alguns métodos possíveis para isso, mas os principais seriam uma comunicação entre processos, retornar o valor/bloco no fim do processo ou acessar usando memória compartilhada. Mas, a comunicação entre processos e o uso memória compartilhada aumenta bastante a complexidade de implementação.

Neste momento não será usada nenhuma forma de implementação de memória compartilhada nessas estratégias.

Diante dessas dificuldades, serão implementadas e comparadas as seguintes estratégias em Python:

- Para cada canal de cor, cada processo irá processar um elemento da matriz de entrada de forma dinâmica e irá retornar cada valor para o processo principal criar a imagem final.

  Primeiramente, devemos obter os dados da imagem a ser processada e, em seguida, com a criação do pool de processos, para cada canal de cor (c), é realizada a convolução para os pixels da imagem de forma paralela. Os valores resultantes da convolução para cada canal são armazenados na imagem de saída de acordo com o retorno de cada processo.
- Para cada canal de cor, dividir a tarefa em blocos de linha/coluna da matriz de entrada de forma igual para cada processo.

  Durante o processo, primeiramente, informações sobre a imagem são obtidas. A imagem é dividida em blocos horizontais, ou seja, o valor da altura da imagem é particionada. Se o número de linhas não for divisível pelo número de processos, o último bloco inclui todas as linhas restantes. Um pool de processos é criado com o número especificado de processos. Para cada canal da imagem, cada processo executa a convolução para seus respectivos blocos. Os resultados dos blocos convoluídos são concatenados verticalmente para reconstruir a imagem final para cada canal.
- Para fins de comparação, será usada a Biblioteca Numba para gerar uma solução concorrente. Numba traduz as funções nativas de Python para código de máquina otimizados em runtime usando o compilador LLVM, fazendo com que o código deixe de ser interpretado nativamente pelo Python, o que torna algoritmos numéricos como este aproximar uma velocidade similar ao C ou FORTRAN.

  Será usado o decorador chamado `@jit` que faz essa tradução para código de máquina, junto com o parâmetro `"parallel=True"`. Este parâmetro paraleliza automaticamente funções que a biblioteca julga ser paralelizável usando Threads e o executa otimizando a cache. Além disso, foi feito uma marcação em cada loop de forma explicita usando prange para ser forcado a paralelização neles e limitando o número de threads de acordo com a entrada do usuário.

2.2. C

Diferente de Python, a linguagem C não possui limitações que restringem o uso de threads para este problema. Assim, para a solução concorrente, foi implementada a divisão por blocos, da mesma forma que usamos em Python, sendo que esta estratégia nos permite que os acessos à memória sejam mais sequenciais, aproveitando melhor a cache. Além disso, não seria necessário uma estratégia dinâmica, como uma bolsa de tarefas, já que cada cálculo da convolução tem basicamente o mesmo tempo de processamento.

A implementação em C realiza a convolução para somente um canal da imagem e precisa receber como entrada os arquivos binários da matriz de entrada (imagem) e do kernel para retornar a matriz convolucionada no formato binário. Isso significa que se quisermos convolucionar uma imagem colorida (com 3 canais), temos que executar o programa 3 vezes: uma para cada canal como arquivo binário de entrada e juntar depois

para recriar a imagem. Além disso, o programa precisa receber o numero de threads a ser criada, caso seja igual a um, será feito de forma sequencial.

Para recriar a imagem original, o programa de Python possui diversas funções que criam vários arquivos binários a partir de uma imagem ou de um kernel em específico para ser usado em C. Há também funções para recriar a imagem usando a matriz em formato binário.

3. Casos de teste

Para testar os programas, foi feito uma classe em Python em que gera testes de forma automática tanto para o programa em Python quanto para C em todas as implementações das estratégias acima.

Tal classe é responsável por criar uma série de imagens antes de iniciar todos os testes, com valores aleatórios para os pixels, em escala de cinza (1 canal) e em cores (3 canais) para os tamanhos:

- 333x333
- 500x500
- 1024x768

Já o kernel escolhido para todos os testes de corretude é o de detecção de bordas :

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

O teste de corretude é feito da seguinte forma:

3.1. Python

Para a versão em Python serão feitos dois testes de corretude, um para imagens em escala de cinza e outro em cores. Em cada um deles são testadas as imagens geradas anteriormente, de acordo com o seu número de canais. Cada imagem será testada com 1, 3, 7, 20 processos (threads no caso do Numba) para testar com um número variado de processos a fim de garantir seu funcionamento em diversas condições.

Para cada imagem e seu número respectivo de processos, é realizado um `AssertEqual` em cada valor da matriz de saída entre sua versão sequencial e a concorrente para garantir que é a mesma saída.

3.2. C

Já em C serão convertidas todas as imagens de teste e o kernel escolhido em arquivos binários para o uso do programa em C. Após isso, a partir código fonte, será gerado o executável com o GCC e o executa criando um processo pelo próprio Python com os parâmetros definidos em cada teste.

Da mesma forma como foi feito em Python, cada imagem será testada com 1, 3, 7 e 20 threads e será feito um `AssertEqual` em cada valor da matriz de saída entre sua versão sequencial e a concorrente para garantir que é a mesma saída.

3.2.1. Garantia da mesma matriz saída entre Python e C

O último teste garante que as saídas do programa em C e Python são a mesma para garantir uma comparação justa entre eles. Nesse contexto é feito o mesmo processo de corretude de C, só que será comparado apenas a versão sequencial de C e Python usando o `AssertEqual` na matriz de saída de cada canal.

4. Avaliação de desempenho

Para a avaliação de desempenho, a mesma classe de teste possui os testes de desempenho para cada linguagem, que armazena o tempo de processamento em um arquivo csv. O tempo de processamento é calculado considerando apenas o tempo de processar a matriz, desconsiderando o tempo de inicialização e de finalização, ou seja, para cada estratégia de implementação só será contabilizado o tempo total para a função de convolução na matriz inicial, tanto em C quanto em Python.

Para isso, serão criado três imagens em escala de cinza e seus respectivos arquivos binários que serão sempre usadas para todos os testes de desempenho com dimensões:

- 500x500
- 1000x1000
- 5000x5000

4.1. Metodologia

Cada imagem será testada com a respectiva solução sequencial e com 1, 2, 4 e 8 processos/threads para cada linguagem. Tais cálculos serão feitos em um MacBook Air M2 de 8 GB de memória, que possui 8 núcleos - sendo 4 deles de eficiência - com um clock de 3.48 GHz cinco vezes para cada teste para calcular sua média e desvio padrão para fazer os cálculos, sendo o desvio padrão representado pela área marcada ao redor da marcação da média na matriz - linha tracejada.

Após os cálculos, serão mostrados o gráfico de Aceleração e Eficiência.

4.2. Python

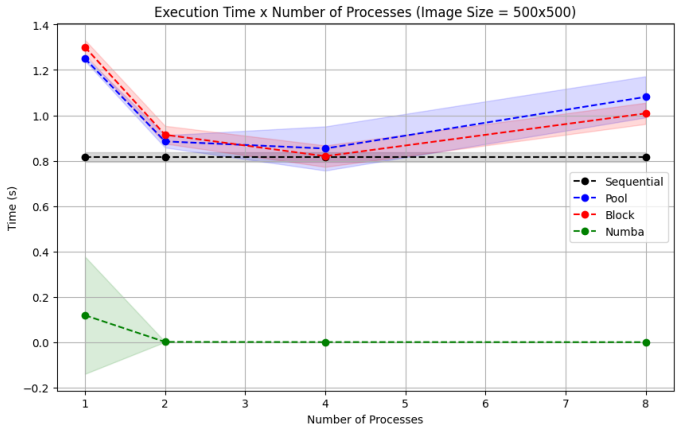

Figure 3. Tempo de Execução em Python (500x500).

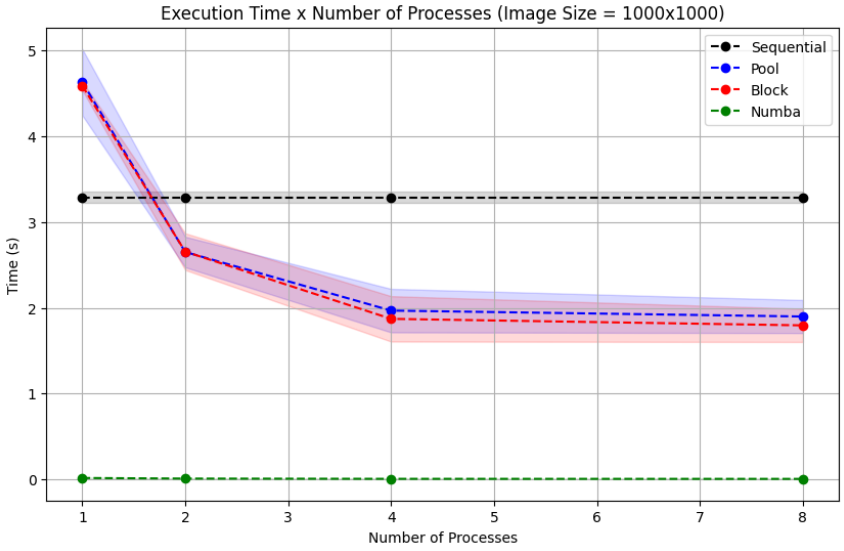

Figure 4. Tempo de Execução em Python (1000x1000).

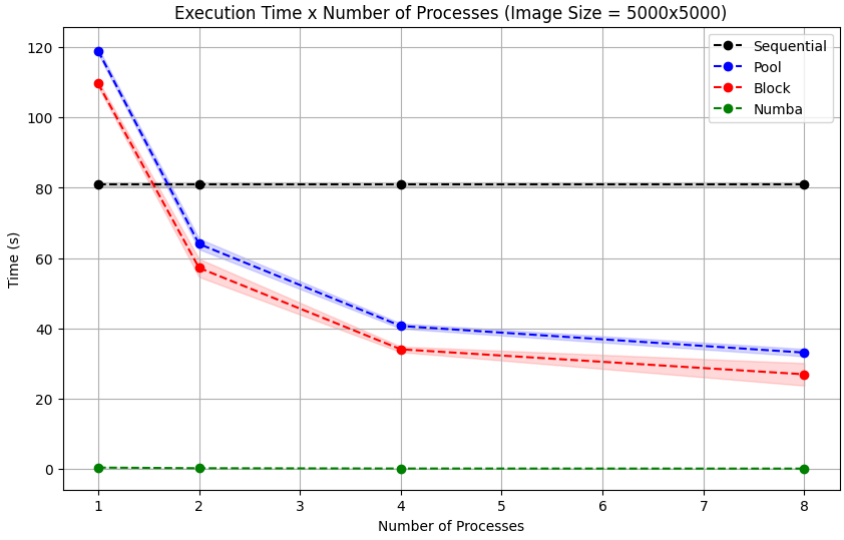

Figure 5. Tempo de Execução em Python (5000x5000).

Como podemos perceber no gráfico, houve pouca variância nos dados coletados, como mostra a área colorida, que é bem sucinta. Também é visível a diferença entre a implementação com Numba e as outras. Por conta disso, nos gráficos de aceleração e eficiência serão omitidos a implementação com Numba pois será feito uma comparação mais específica entre Numba, C e o Python em uma escala mais amigável mais tarde.

Outro ponto importante é que é perceptível a diferença no desempenho entre as duas estratégias de divisão, em que a divisão de blocos possui uma performance melhor em quase todos os casos em relação à estratégia usando a divisão dinâmica de tarefas com o pool. Muito provavelmente isso se deve por conta da maior eficiência da cache e por retornar de uma vez os resultados ao processo principal.

4.2.1. Aceleração

O calculo da Aceleração está sendo feita com base no tempo sequencial de Python em relação à média dos tempos de processamento. Assim, o cálculo da aceleração é a razão entre o tempo de execução da versão sequencial do algoritmo ($Ts(n)$) e o tempo de execução da versão concorrente ($Tp(n, p)$) usando p processadores - ou processos, neste caso, já que processos é igual ao número de processadores usado.

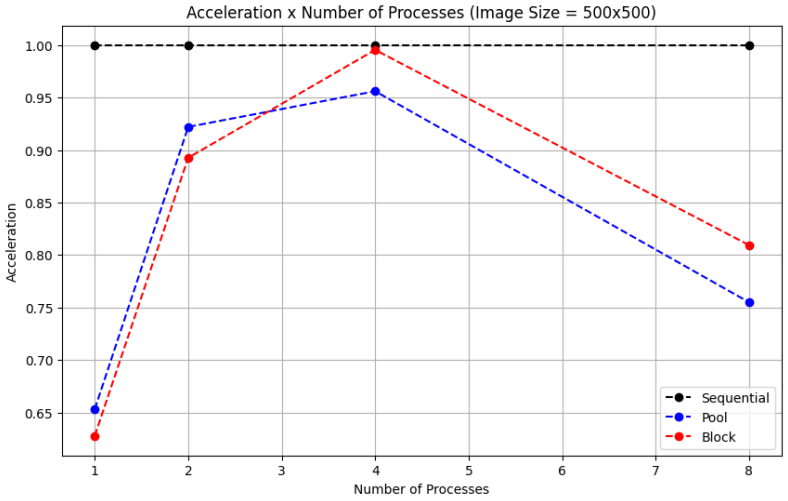

Figure 6. Aceleração em Python (500x500).

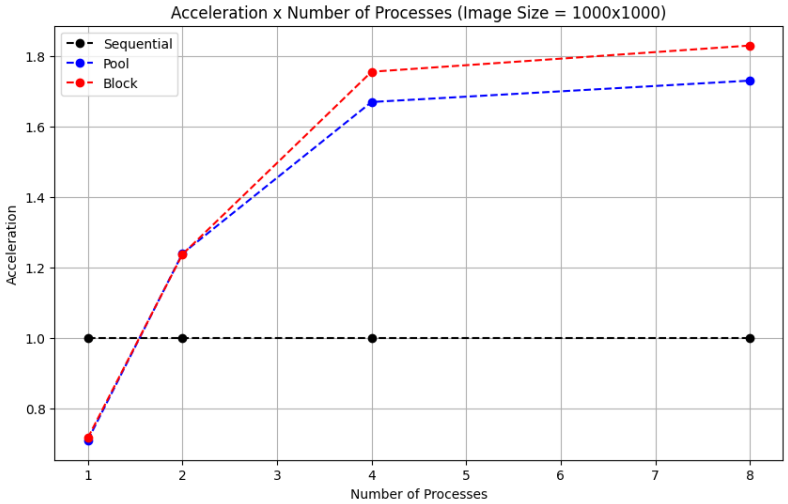

Figure 7. Aceleração em Python (1000x1000).

Um caso interessante que foi visto nestes testes com o Python é a questão da aceleração em entradas muito pequenas. Neste caso, nenhuma solução concorrente conseguiu ser melhor que a sequencial. Isso se deve por conta do ”overhead” da criação processos, o que impacta na aceleração e eficiência do código, em que em uma imagem pequena (500x500) possui um rendimento pior em todos os casos de teste, somente havendo uma melhora no desempenho quando a imagem é relativamente grande.

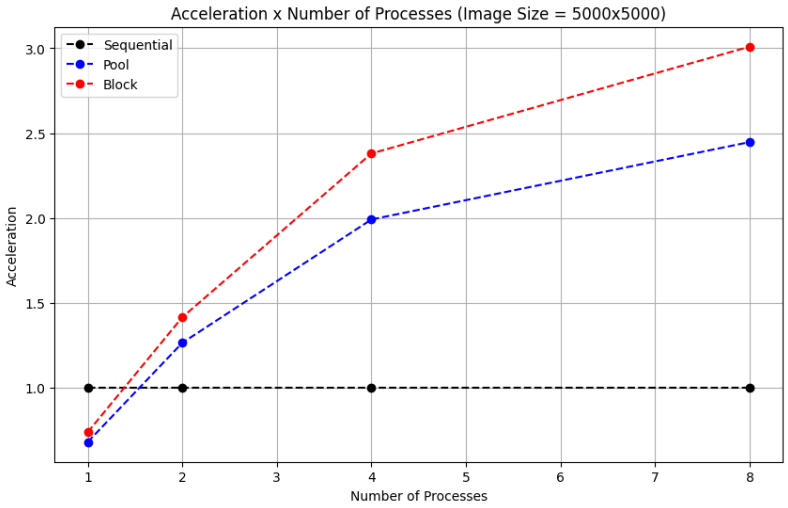

Figure 8. Aceleração em Python (5000x5000).

4.2.2. Eficiência

Já para o valor da eficiência está sendo calculado a razão entre a aceleração e o número de processadores usado.

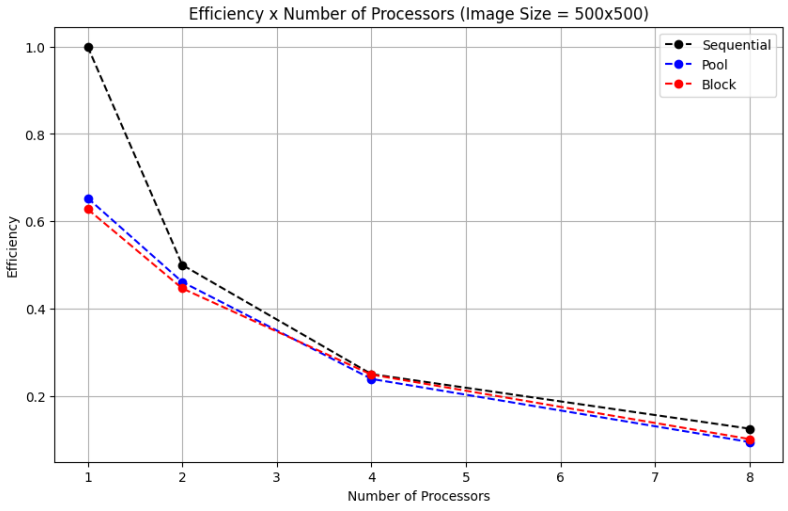

Figure 9. Eficiência em Python (500x500).

Uma questão importante de salientar é o fato da eficiência de Python, junto com a aceleração, estar abaixo do esperado. Além das particularidades encontradas quando a imagem é pequena, vemos que a eficiência decai bastante com o uso de processadores, ficando apenas um pouco acima da eficiência da implementação sequencial.

Além disso, a eficiência aumenta de acordo com o tamanho da imagem, como é visto comparando a figura 10 e 11, e a discrepância entre as duas implementações concorrentes aumenta ainda mais, mostrando a superioridade da estratégia de divisão em blocos.

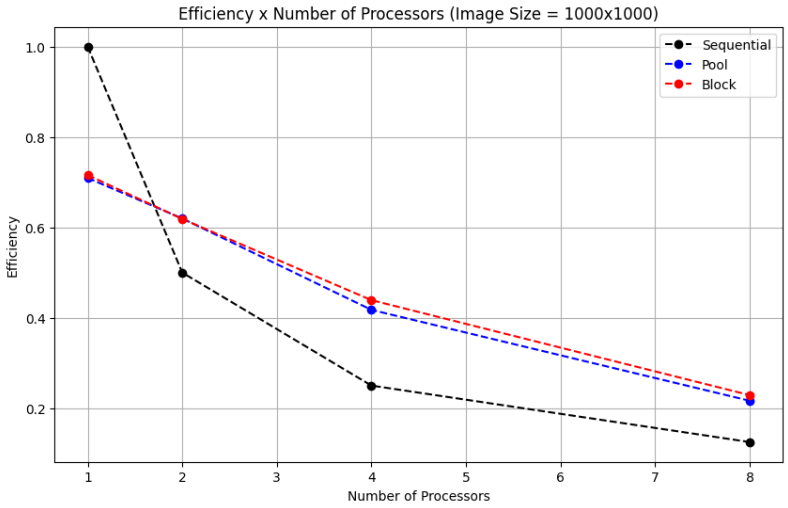

Figure 10. Eficiência em Python (1000x1000).

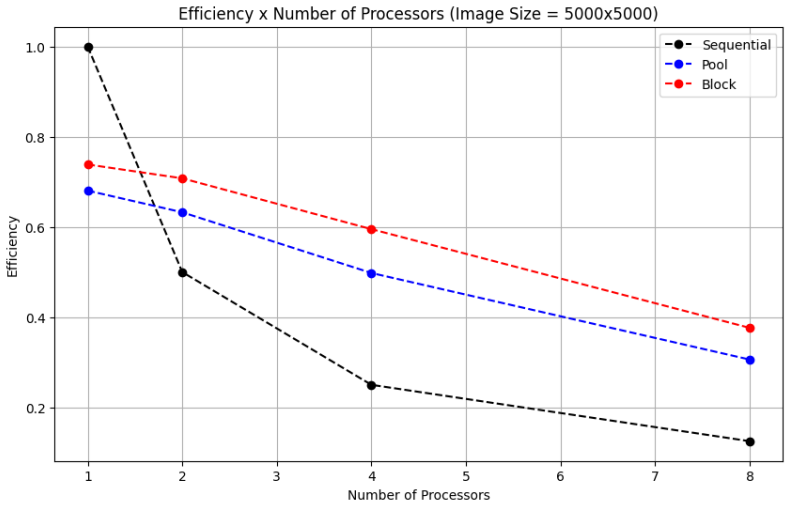

Figure 11. Eficiência em Python (5000x5000).

4.3. C

Como podemos perceber, a implementação usando a linguagem C é muito mais rápida que em Python em todos os testes. Muito disso se deve por conta das particularidades de Python, em que é uma linguagem interpretada em tempo real (runtime) do que compilada direto em código de máquina. Por conta desse motivo, o código está quase cem vezes mais rápido apenas comparando suas versões sequenciais.

Outro ponto importante a mencionar é que nos gráficos a seguir, a versão sequencial em C está sendo mostrada como o tempo de uma thread, os outros números são os tempos concorrentes em seus respectivos números de thread. Além disso, visível a baixa variância entre os dados coletados.

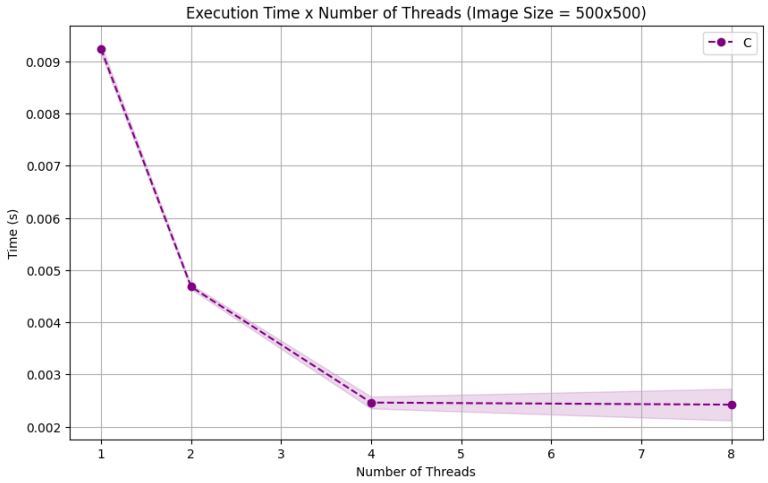

Figure 12. Tempo de Execução em C (500x500).

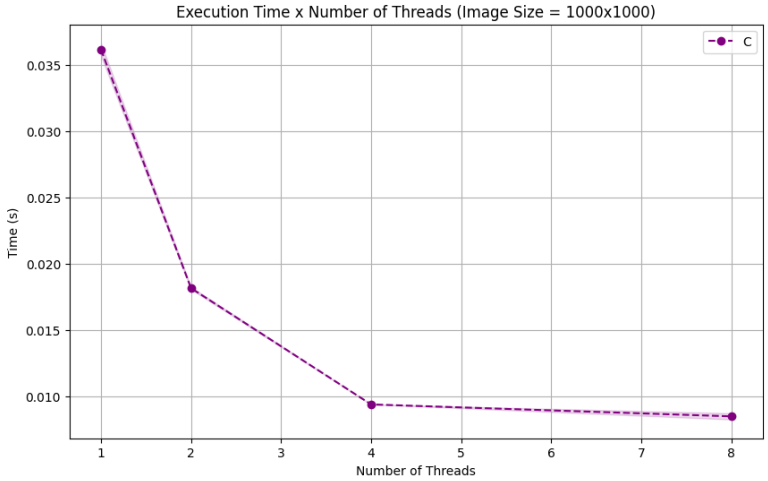

Figure 13. Tempo de Execução em C (1000x1000).

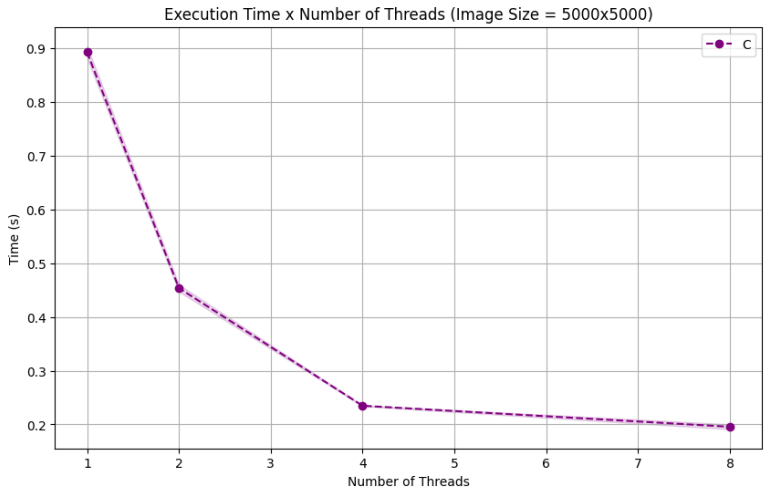

Figure 14. Tempo de Execução em C (5000x5000).

4.3.1. Aceleração

O calculo da Aceleração está sendo feita com base no tempo sequencial de C em relação à média dos tempos de processamento, seguindo a mesma fórmula anterior.

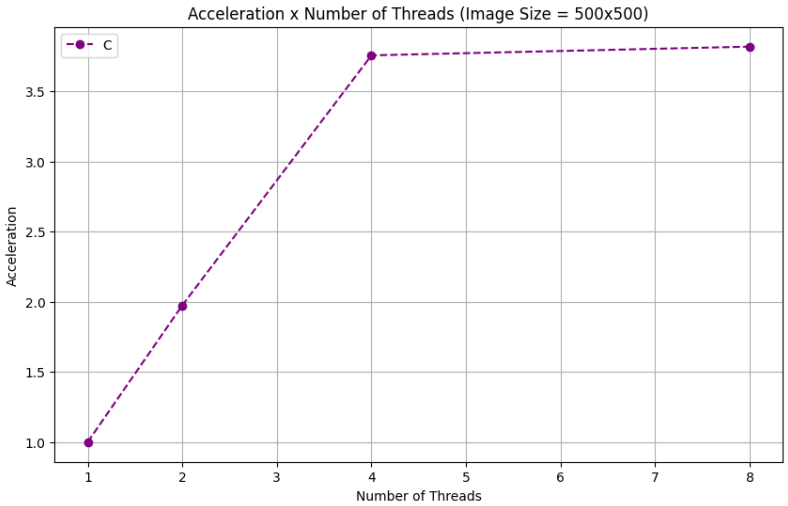

Figure 15. Aceleração em C (500x500).

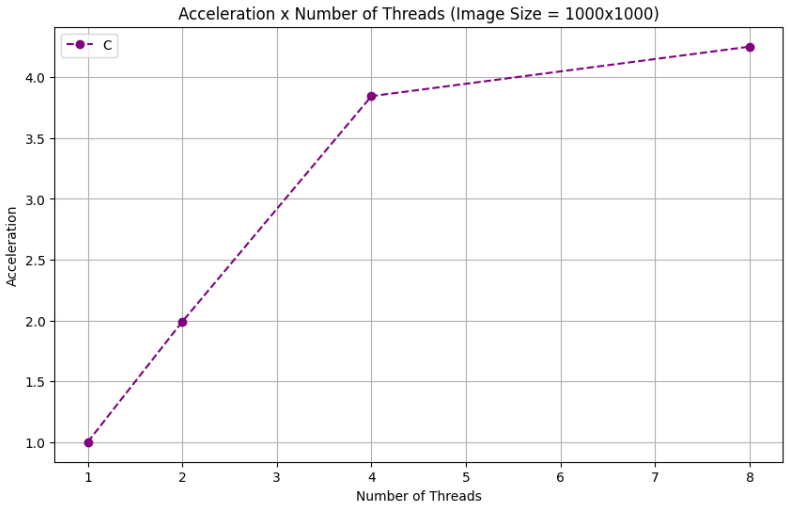

Figure 16. Aceleração em C (1000x1000).

Uma diferença gritante entre a aceleração entre Python e C é que, independente do tamanho da matriz de entrada, sua aceleração é quase linear. Dessa forma, atingimos quase o ideal em um algoritmo concorrente, em que com p processadores o algoritmo concorrente executaria p vezes mais rápido que o algoritmo sequencial, o que teoricamente esse é o valor máximo de aceleração de um algoritmo.

Apesar disso, é visível que a aceleração decai substancialmente após 4 threads - o mesmo que o número de processadores neste caso - por conta dos processadores de eficiência da maquina utilizada para os testes já que o sistema operacional - no caso, MacOS (baseado em UNIX) - escalona a thread para esses núcleos que são mais fracos em termos de performance.

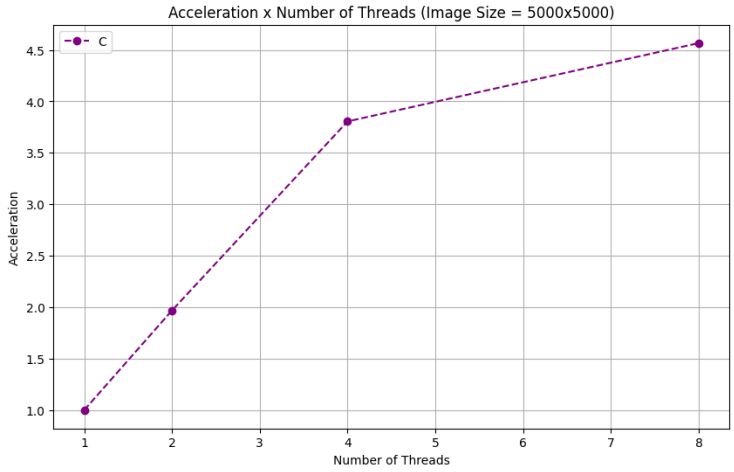

Figure 17. Aceleração em C (5000x5000).

4.3.2. Eficiência

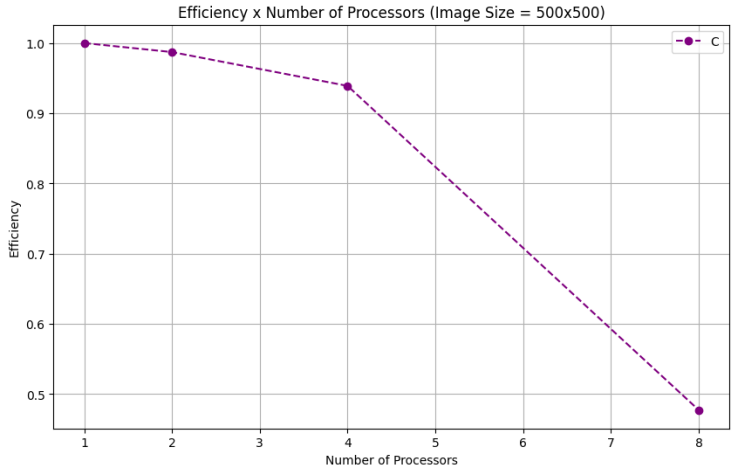

Figure 18. Eficiência em C (500x500).

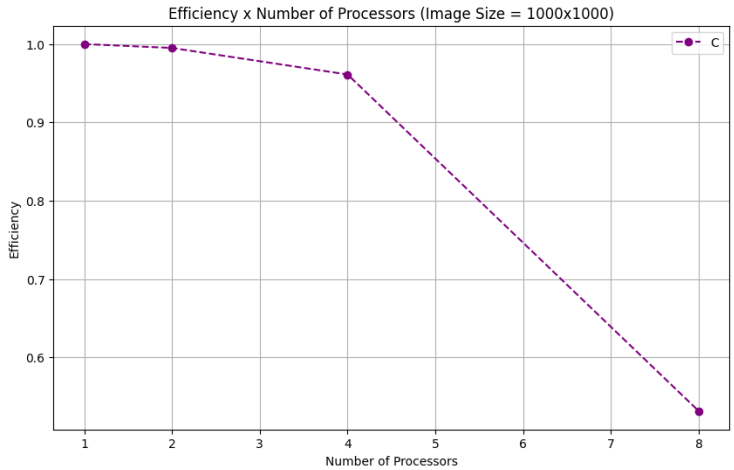

Figure 19. Eficiência em C (1000x1000).

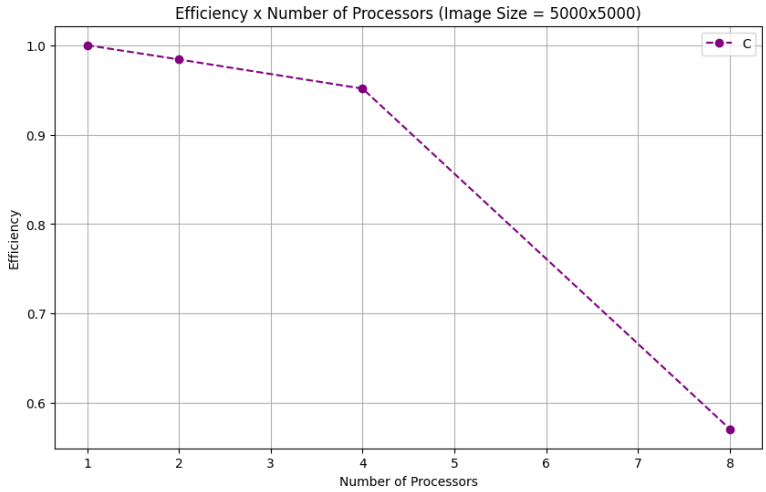

Figure 20. Eficiência em C (5000x5000).

Por fim, podemos perceber a queda brusca de eficiência a parir de 4 threads, como foi explicado anteriormente. Mas mesmo assim, a eficiência em C é muito superior ao de Python quando comparamos seu ganho em relação ao sequencial de cada implementação. Podemos perceber que em C se mantém bem próximo de 1 no início, enquanto em Python já decai para quase 0.6.

4.4. C x Python x Numba

Para comparar melhor a implementação com Numba em relação aos demais, todas as escalas estão feitas de forma logarítmica em relação ao tempo por conta da gritante diferença entre os tempos.

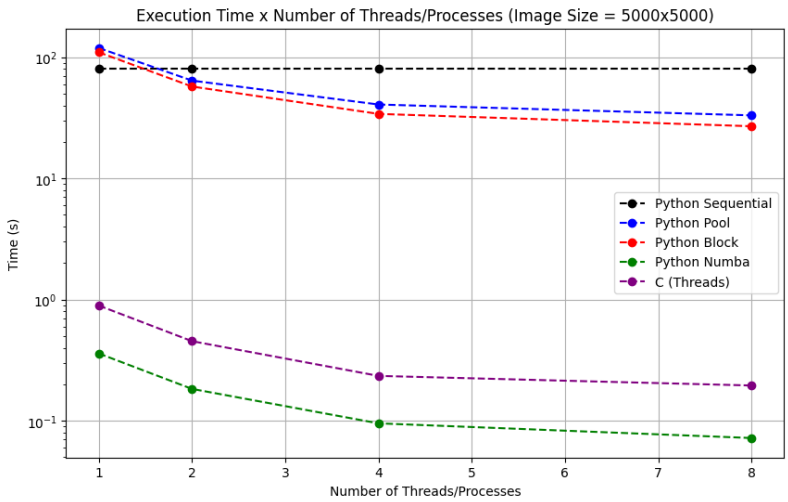

Figure 21. C x Python - Escala Logarítmica (5000x5000)

Impressionantemente, a versão com Numba consegue ser mais rápida do que em C em todos os casos de teste de maneira significativa. Além disso, nesse gráfico fica evidente o quão lento as implementações usando nativamente o Python é em relação à C e ao Numba.

4.4.1. Aceleração

O cálculo da aceleração continua a mesma das anteriores, porém a do Numba(Python) é feita em relação à versão sequencial em Python. Nesse caso, por conta dos cálculos

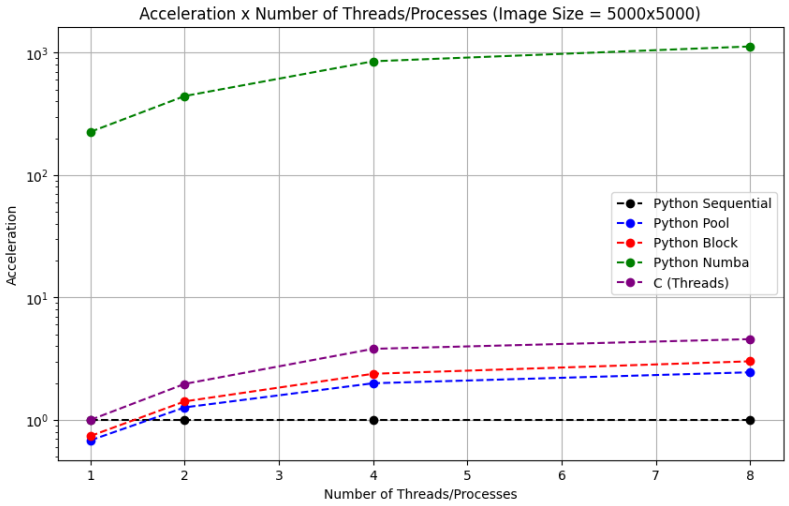

Figure 22. Aceleração C x Python - Escala Logarítmica (5000x5000)

acima, a versão do Numba teve uma aceleração muito grande às demais implementações em Python, o que enfatiza o ganho de desempenho ao se usar esta biblioteca caso esteja fazendo algum tipo de algoritmo numérico em Python. O C neste caso está muito abaixo pois seu cálculo não é baseado na versão sequencial em Python, mas sim sua própria em C.

4.4.2. Eficiência

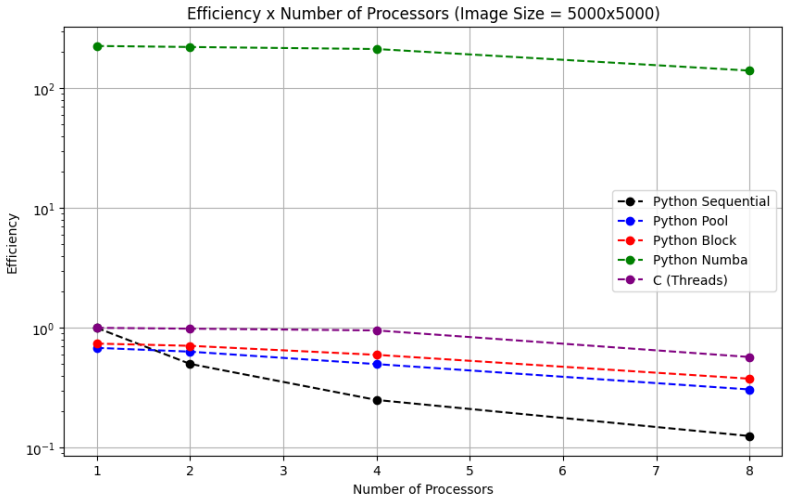

Figure 23. Eficiência C x Python - Escala Logarítmica (5000x5000)

5. Discussão

Inicialmente, o projeto se pautou em implementar uma versão concorrente de um algoritmo de convolução de imagens em Python. Porém, ao decorrer da implementação, foi visto um alto tempo para completar uma simples convolução e assim foi feito um teste para verificar como seria o tempo em uma linguagem como C. Com essa curiosidade, veio a descoberta da diferença entre o tempo de processamento entre as linguagens, que gerou uma grande pesquisa, gerando esse relatório.

Assim, buscando alternativas para resolver a lentidão em Python, implementamos uma solução usando uma biblioteca chamada Numba que, como vimos acima, superou todas as expectativas, conseguindo uma performance superior à implementação em C em todos os testes. Sendo esta a melhor opção de implementação, com uma eficiência e aceleração muito superiores à C e com a vantagem de manter a simplicidade de Python.

Após terminar esse relatório e analisar as métricas de desempenho em conjunto, é possível perceber o quão lento o Python é em relação ao C, sendo aproximadamente cem vezes mais lento. Tal resultado foi muito pior que o esperado, o que demonstra que é muito mais vantajoso usar bibliotecas como o Numba em casos em problemas numéricos que usam bastante poder computacional de forma paralela ou até mesmo implementar o mesmo problema em uma linguagem mais eficiente para ganhos de performance, não só buscar uma implementação concorrente.

Apesar do projeto ser bastante simples, a implementação em Python com processos foi um empecilho. Um incremento interessante ao projeto seria tentar implementar novamente uma memória compartilhada entre os processos para salvar o resultado da matriz final, já que não conseguimos implementar nesse projeto por conta de diversos erros em Python, o que mostra sua dificuldade de implementação. Com isso, comparar sua performance em relação às demais estratégias, apesar de que, muito provavelmente, não iria superar as velocidades de Numba ou C.

Outro ponto que poderia melhorar seria a implementação de leitura e escrita de imagens diretamente no programa em C, sem necessitar de arquivos binários.

Por fim, queríamos agradecer bastante à professora Silvana pelas aulas e pela oportunidade de desenvolver este trabalho.

Bibliografia

[1] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[2] Cristóbal A. Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15(2):285–329, 2014.

[3] Setosa. Image kernels, 2023. Accessed: 2024-06-13.

[4] Vincent Mazet. Convolution, 2023. Accessed: 2024-06-13.

[5] Demo Fox. Image sharpening convolution kernels, 2022. Accessed: 2024-06-13.

[6] Shuling Yu, Quinn Snell, and Bryan Morse. Parallel algorithms for image convolution. 03 1998.

[7] Numba. Numba docs, 2024. Accessed: 2024-06-13.