Teste Modelo 2

```
int** blurImage(int image[N][N]) {
    int temp[N][N];

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            int sum = 0, count = 0;
            for (int di = -1; di <= 1; di++) {
                for (int dj = -1; dj <= 1; dj++) {
                    int ni = i + di, nj = j + dj;
                    if (ni >= 0 && ni < N && nj >= 0 && nj < N) {
                        sum += image[ni][nj];
                        count++;
                    }
                }
            }
            temp[i][j] = sum / count;
        }
    }
    return temp;
}

void combineImages(int image1[N][N], int image2[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            image1[i][j] = (image1[i][j] + image1[i][j - 1] + image2[i][j]) / 3;
        }
    }
}

int* createHistogram(int image[N][N]) {
    int histogram[256];
    for (int i = 0; i < 256; i++) histogram[i] = 0;

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            histogram[image[i][j]]++;
        }
    }
    return histogram;
}

int main() {
    // ... read image1 and image2 ...

    image1 = blurImage(image1);           // Apply blur filter on image1
    combineImages(image1, image2);        // Combine image1 with image2
    int* histogram = createHistogram(image1);  // Create a histogram

    // ... save image1 and histogram ...
}
```

1.Sequential Program Analysis and Optimization Pretende-se analisar e otimizar a versão sequencial deste programa.

a) 15 pts Indique a complexidade de cada uma das funções invocadas na main.

tudo O(N^2)

b) 15 pts Apresente duas otimizações possíveis e discuta o seu impacto no tempo de execução com base na equação de desempenho do processador

Acho que as matrizes podiam estar guardadas de uma forma mais eficiente, ou com um tipo de dados mais pequeno que inteiro ou em forma de linha.

Acho que podíamos proceder a loop unrolling.

2.

a) 30 pts Indique, justificando, se a função combineImages apresenta bloqueadores de paralelismo. Discuta modificações ao código para remover eventuais bloqueadores.

Sim, o cálculo para cada célula depende da célula na mesma coluna, na linha acima, o que obriga a que as operações numa coluna sejam feitas sequencialmente. Podemos optar por separar em dois ciclos, um com a parte dependente e outro sem.

```
image1[i][j] = (image1[i][j] + image1[i][j - 1] + image2[i][j]) / 3; é igual a
image1[i][j] = (image1[i][j] + image2[i][j]) / 3 + image1[i][j - 1] / 3;
```

```
#pragma omp parallel for collapse (2)
    for(int i = 0; i < N; i++){
      for(int j = 1; j < N; j++) {
        image1[i][j] = (image1[i][j] + image2[i][j])/3;
      }
    }
```

```
#pragma omp parallel for
    for(int i = 0; i < N; i++){
      for(int j = 1; j < N; j++) {
        image1[i][j] = image1[i][j] + image1[i][j-1]/3;
      }
    }
```

b) 15 pts Desenvolva a versão paralela das funções blurImage, e comente o propósito de cada uma das diretivas que usar neste contexto.

usaria omp for, que repartiria as iterações do ciclo “i” por um grupo de threads de igual forma. Podia ainda acrescentar o collapse(2), mas não surtiria grande efeito dado que a matriz é quadrada. Poderia também reafirmar as variáveis privadas e partilhadas com o private(...) e shared(), mas estas estão implicitamente expostas.

3. CUDA Kernel Implementation

```
__global__ void blurImageKernel(int **image) {  // Image is a 2D array

    int thread_id_x = threadIdx.x + blockIdx.x * blockDim.x;

    int thread_id_y = threadIdx.y + blockIdx.y * blockDim.y;

      int i = thread_id_x;
      int j = thread_id_y;
```

```
            int sum = 0, count = 0;
            if (i < N && j < N ) {
            for (int di = -1; di <= 1; di++) {

                for (int dj = -1; dj <= 1; dj++) {
                    int ni = i + di, nj = j + dj;

                    if (ni >= 0 && ni < N && nj >= 0 && nj < N) {
                        sum += image[ni][nj];
                        count++;
                    }
                }
            }
            temp[i][j] = sum / count;
        }
}
```

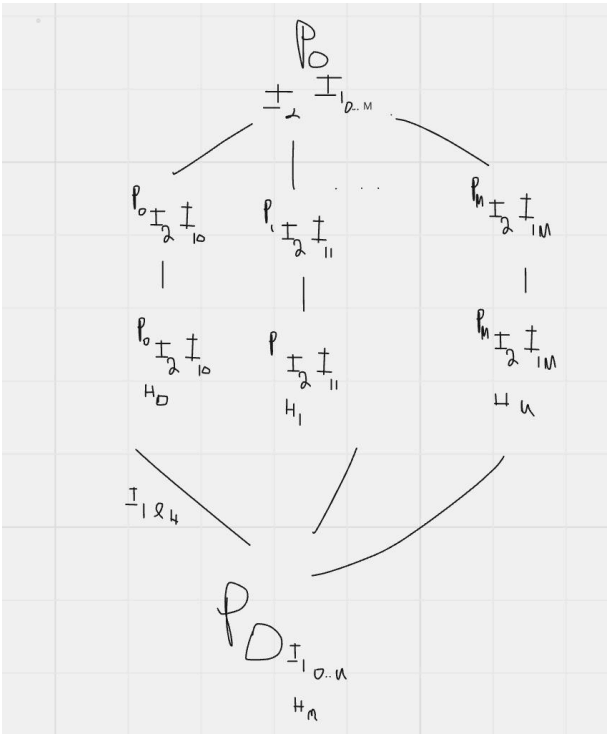
Uma thread por pixel se possivel!

4- Distributed Memory with MPI

Pretende-se desenvolver uma versão paralela do código para memória distribuída com passagem de mensagens (i.e., MPI) em que o processo 0 contém M imagens (várias image1) e, no fim, deverá receber as imagens processadas.

a) 15 pts Identifique, justificando, qual o padrão que se adequa melhor à resolução do problema com MPI, a partir dos padrões paralelos pipeline e farm. Apresente um diagrama com as trocas de mensagens necessárias entre os vários processos, para o padrão escolhido (ilustre com processos).

No contexto deste problema, temos várias imagens que precisam de ser processadas, mais uma imagem que servirá como base para o processamento das M. O processamento inclui essencialmente três etapas sequenciais: Aplicar blur na imagem1, combinar imagem1 com a imagem2 e criar um histograma. Não existe um fluxo contínuo de dados onde um “pipeline” (linhas de produção de tarefas) seja a melhor solução. Em contrapartida, temos um único grande conjunto de dados e operações homogéneas sobre diferentes partes da imagem, o que se adapta melhor a um padrão do tipo “farm” (ou “master-worker”), no qual:



b) 15 pts Indique, justificando, as diretivas de comunicação a usar para implementar o padrão de comunicação proposto na alínea anterior.

As diretivas de comunicação seria o “Scatter” + o “Brodcast” para os histogramas e imagem 2. No caso das Imanges1, seria preciso enviar uma a uma, pois no enunciado não é referido que as imagens são de dimensões iguais (Mpi só permite scatter de arrays lineares!). Se considerarmos que as combinações de Imagem1 com Imagem2 serve como restrição forte o suficiente para os tamanhos das Imagens 1 serem iguais, então podemos usar scatter. De seguida, recebemos as imagens1 no p0 (Receive ou Scatter depende) e finalmente “Gather” dos histogramas para o p0

Paralelismo ao nível da instrução (20p)

1. Indicar as otimizações ILP efetuadas pelo hardware. (10p)

Dependendo do hardware, existem diferentes opções de otimização ao nivel da instrução. Alguns exemplos de otimizações é o pipe-lining, o que permite que diversas instruções executem ao mesmo tempo, execução fora de ordem, o processador reorganiza dinamicamente a ordem de execução das instruções para minimizar dependências e especulação, onde o processador executa instruções antes de confirmar se elas realmente serão necessárias (ex.: previsões de desvios).

2. As otimizações ILP podem ser comprometidas por dependências entre as instruções. Indicar quais são, dando um exemplo (e.g. código assembly, pseudocode) para cada. (10p)

Duas dependencias estudadas são as seguintes:

RAW (Read after Write)

mov 0, %r10

add %r10, %r11

Dependencia de controlo

If A then B else C, neste caso só depois de A sabemos que codigo executar

2. SIMD: explicar em que consiste o conceito, e em que condições podem ser efetuadas automaticamente pelos compiladores. (10p)

SIMD corresponde a uma série de operações sobre multiplos dados numa só instrução, e também é conhecido com vetorização. Primeiramente, o processador deve ser capaz de sustentar este tipo de operações. De seguida, resta ao compilador compreender se uma secção de código poderá vetorizar ou não. Casos onde não é possível ocorrer vetorização são, por exemplo, iterações de um loop onde existem dependecias de dados ou if then elses. Para permitir a vetorização, devemos maximizar a localidade espacial dos dados, melhorando se possivel o seu alinhamento na memória, como caso de AoS e SoA.

3. Explicar a diferença entre os conceitos de Programa, Processo, e Fio de execução (Thread). Explicar qual o efeito dos comandos "export OMP NUM THREADS=10; ./a.out" e "mpi run -np 4 ./a.out" com base nesses conceitos. (20p)

   Programa -> Um programa é um conjunto de instruções ou código armazenado num arquivo que descreve como realizar uma tarefa ou resolver um problema.

   Processo -> Instancia de um programa em execução.Usado para tarefas distintas, espaço de enderaçamento proprio e trocam a nivél do kernel. Derivam de um mesmo programa e podem recorrer a fios de execução.

   Fio -> Fazem parte do mesmo trabalho, partilham espaço de endereços, codigo dados e ficheiros, e também podem trocar tanto a nivél de kernel como a nivél do utilizador. São limitadas por dependencias no programa (e pelos algoritmos)

Diferenças principais: Espaço de Endereçamento: Processos possuem espaço de endereçamento independente, enquanto threads compartilham o mesmo espaço de endereçamento dentro de um processo.

Isolamento: Processos são isolados entre si, enquanto threads de um mesmo processo podem acessar os mesmos dados.

Custo de Troca: Trocar entre processos envolve mais overhead porque requer intervenção do kernel, enquanto a troca entre threads de um mesmo processo é geralmente mais rápida.

4. Comentar a utilização de memórias HBM (High Bandwidth Memory) como solução face ao problema da memory wall, dando ênfase à sua utilização em arquiteturas com vários cores. (10p)

   Melhoria do Paralelismo:

   Em arquiteturas com muitos núcleos, como GPUs modernas ou CPUs multicore massivamente paralelas, a alta largura de banda da HBM permite que múltiplos núcleos acessem os dados sem congestionamento. A arquitetura de canais independentes da HBM é ideal para lidar com acessos concorrentes e equilibrar a carga entre núcleos.

   Redução do Problema de "Cache Miss":

   Com mais dados disponíveis em menos tempo, o impacto de cache misses nos níveis superiores da hierarquia de memória é reduzido, melhorando a eficiência dos núcleos.

   Integração Ideal para GPUs e Computação Acelerada:

   GPUs modernas, como as utilizadas em aplicações de IA e simulações científicas, dependem fortemente de acesso rápido e paralelo à memória. A utilização de HBM elimina gargalos típicos encontrados com memórias DDR tradicionais.

   Escalabilidade para Workloads Intensivos:

Aplicações como aprendizado profundo, modelagem molecular e análise de big data exigem acesso rápido a grandes volumes de dados. A HBM fornece a largura de banda necessária para essas cargas de trabalho.

5. O Top 500 tem duas métricas de avaliação principais: Rpeak e Rmax. Explicar em que consistem as duas e porque é que há sempre uma grande diferença entre ambas. (10p)

O Rpeak representa o desempenho teórico máximo que um supercomputador pode alcançar, baseado nas especificações de hardware, como o número de núcleos, a frequência de operação, e o número de operações de ponto flutuante que cada unidade de processamento pode realizar por ciclo.

O Rmax representa o desempenho real e mensurável, que é sempre menor que Rpeak devido a ineficiências práticas. Mede o desempenho real alcançado pelo supercomputador ao rodar benchmarks como o Linpack. Rmax é obtido executando o problema de sistema linear Ax=bAx=b no supercomputador, usando uma matriz densa. Mede a eficiência real do sistema considerando:

Comunicação entre nós. Limitações de memória. Eficiência do software.