Universidade Federal de São Carlos
Departamento de Computação

Programação Paralela

ARQUITETURAS E ESTRATÉGIAS DE ACELERAÇÃO

Aluno: **Gustavo de Macena Barreto**

Prof. Ricardo Menotti

Prof. Hélio Crestana Guardia

São Carlos, 9 de abril de 2023

1 Descrição do problema

Neste algoritmo, temos como objetivo realizar o cálculo de K-ésimos vizinhos próximos cujo intuito é identificar a categoria A qual este ponto se identifica. Esse tipo de algoritmo[1] é muito utilizado em redes neurais e sistemas de grafo, Nos quais é necessário realizar uma classificação genérica para esse tipo. Dessa forma, foi escolhido como base esse tipo de algoritmo por conta de sua aplicação. Seu funcionamento é relativamente simples, Tem-se como entrada: "n"pontos de classificação conhecida e por fim o ponto desejado classificar.

2 Estratégia de paralelização

Para desenvolvimento, foi inicialmente desenvolvido um algoritmo em OpenMP, no qual se tem maior facilidade e controle por meio de diretivas de compilação e critérios para separação de variáveis que podem ser utilizadas ao longo da paralelização do algoritmo. Além disso sua programação não é mais acessível e direta em relação à programação por meio de *kernels* (CUDA, openCL, SYCL e outras). Com isso, o resultado de programação não foi satisfatório tendo em vista as dependências durante a execução da ordenação(qsort), na qual custou muito tempo para a execução. Além disso, outro algoritmo que será mencionado em seguida, desenvolvido em CUDA, sofre do mesmo problema no quesito de ordenação.

A programação em openMP é baseada em pragmas, dos quais são característicos da linguagem. Esta linguagem oferece diversas funções para o gerenciamento de threads, que permite escalonar diversas operações simultâneas para uma dada tarefa, permitindo intensidade de processamento. Na imagem 1, será explicado a seguir a estratégia tomada para a paralelização.

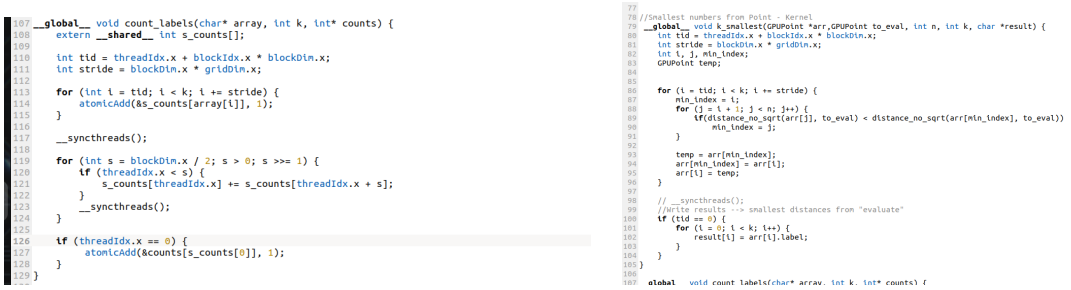
[1] **O material pode ser acessado neste link:** `https://github.com/MBGustav/WSCAD-2022/tree/main/A`

```
char knn_mp(Point *arr, int n , Point P, int k){
    int i, j, min_index;
    Point temp;
    char *result = (char*) malloc(sizeof(char)*k);

    #pragma omp parallel for private(i, j, min_index, temp) shared(arr, result)
    for (i = 0; i < k; i++) {
        min_index = i;
        for (j = i + 1; j < n; j++) {
            if(euclidean_distance_no_sqrt(arr[j], P) < euclidean_distance_no_sqrt(arr[min_index], P))
                min_index = j;
        }

        temp = arr[min_index];
        arr[min_index] = arr[i];
        arr[i] = temp;
    }

    //Write results --> smallest distances from "evaluate"
    #pragma omp barrier
    if (omp_get_thread_num() == 0) {
        for (i = 0; i < k; i++) {
            result[i] = arr[i].label;
        }
    }

    qsort(result, k, sizeof(char), compare_for_sort);

    char most_freq = most_frequent(result, k);
    return most_freq;
}
```

Figura 1: Algoritmo em openMP

É declarado o *parallel* for, no qual o loop principal será realizado de forma paralela, realizando o calculo de distancia entre o ponto P e o i-ésimo ponto com classificação. Vale ressaltar que nesta etapa não há condição de corrida entre as threads, visto que não há acesso concorrente entre as threads. Vale salientar que os vetores *arr* e *result* são elementos compartilhados, justamente para partilhar o acesso de dados simultaneamente entre as *threads*. Finalizada esta execução, é necessário aguardar a execução em sua totalidade a execução das distancias que preenchem os dados de *arr*, pois após este cálculos, haverá dependência entre dados para a transposição da categoria do ponto. Vale ressaltar que essa tranposição é feita justamente para obter-se qual a moda no vetor de char's result.

Nesta parte, a explicação será dividida em duas etapas: wrappers e declaração de kernels. Primeiramente, a declaração de wrappers tem como objetivo a facilitação do uso de kernels por meio da linguagem de paralisação CUDA, isso porque existe a necessidade de configuração previas do envio de kernel, tal como: alocação de memória dinâmica, total de blocos, e número de threads por bloco. Com isso, a distribuição e organização de dados se torna mais evidente ao longo da programação, facilitando assim eventual alteração ou melhorias de código , se necessário.



(a) (b)

Figura 2: Implementação em Cuda - Declaração de Kernels

Na imagem 2(b) e 2(a),é apresentado a declaração por meio da linguagem de programação CUDA. Nessa etapa é realizada a compilação direta para linguagem de programação voltada para placas de vídeos da NVidia. O primeiro algoritmo apresenta um contador de categorias cujo intuito é contabilizar a categoria que mais se repete, tendo em vista a quantidade total daquele elemento, isto é, a moda daquele grupo representado pelo vetor nomeado de "array". Já no segundo, é declarado o Kernel que tem o papel principal de classificar os pontos mais próximos dado o ponto P. Neste inquérnia é realizado de maneira paralela o cálculo de distâncias sem o uso de raízes quadráticas pois isso reduz o tempo de execução e também não será necessário para esse caso. Após o cálculo de distância é preenchido o vetor result, cuja função é acumular os pontos mais próximos na ordem de k, no qual será utilizado pelo primeiro algoritmo mencionado anteriormente.

Conforme mencionado anteriormente, os wrappers tem como objetivo facilitar e evidenciar características na programação na placa de vídeo. É possivel notar funções de transmissão de dados de host para device e vice-versa. Além disso, existe alocações

```
char most_frequent(char *d_array, int k) {
    const int num_labels = 256; // assuming ASCII encoding
    int* d_counts;
    cudaMalloc(&d_counts, num_labels * sizeof(int));
    cudaMemset(d_counts, 0, num_labels * sizeof(int));

    // char* d_array;
    // cudaMalloc(&d_array, k * sizeof(char));
    // cudaMemcpy(d_array, array, k * sizeof(char), cudaMemcpyHostToDevice);

    const int threads_per_block = 256;
    const int num_blocks = (k + threads_per_block - 1) / threads_per_block;
    const int shared_memory_size = num_labels * sizeof(int);

    count_labels<<<num_blocks, threads_per_block, shared_memory_size>>>(d_array, k, d_counts);

    int* h_counts = new int[num_labels];
    cudaMemcpy(h_counts, d_counts, num_labels * sizeof(int), cudaMemcpyDeviceToHost);

    char most_freq = 0;
    int most_freq_count = 0;
    for (int i = 0; i < num_labels; i++) {
        if (h_counts[i] > most_freq_count) {
            most_freq_count = h_counts[i];
            most_freq = (unsigned char) i;
        }
    }

    delete[] h_counts;
    cudaFree(d_counts);
    cudaFree(d_array);

    return most_freq;
}
```

(a) (b)

Figura 3: Implementação em Cuda - Declaração de *Wrappers*

prévias com chamadas de prevenção de erros ao longo desses tipos de chamadas. Vale salientar que esses tipos de checagem de erros, feitos por meio de *checkCudaError*, visam evitar falha de transmissão entre dispositivos.

3 Análise da escalabilidade: esperada e obtida

Apesar da característica e desempenho notável para o cálculo de distâncias de maneira paralela, seja em openMP, seja em cuda, isso não foi o suficiente para obter um resultado satisfatório. A necessidade de ordenar os vetores e consequentemente a redução dos mesmos para um vetor exige a necessidade de um dataset relativamente grande, para que haja ganho de empenho nas plataformas. O principal gargalo observado nesse algoritmo está relacionado com a transmissão e quantidade de dados necessários. Uma possível melhoria para este algoritmo seria a locação de memória em local para melhor manejo dos dados presentes na placa de vídeo.

4 Conclusão

Por fim há de se concluir, e consequente redução dos dados em posições menores corroboraram para um desempenho não tão satisfatório com esperado. Possíveis melhorias podem ser feitas nesses códigos, das quais permitem agregar maior desempenho para o algoritmo, tal como melhor manejo e transmissão de dados, além de ordenar ou refaturar o atual modelo de ordenação desse algoritmo. Além disso seu atual funcionamento se encontra similar a CPU na qual seu funcionamento serial se tornou equivalente com acelerador.