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, 10 de abril de 2023

1 Descrição do problema

Este algoritmo tem como objetivo ou realizar o cálculo de um fractal conhecido como set Júlia[1]. Esse tipo de algoritmo é conhecido pela sua estrutura de recursividade por meio da Cópia reduzida de seu Contorno. Dentre outros fractais muito conhecidos ,existem: Floco de neve de Koch, mandelbrot e outros. Neste projeto será tomado como referência o fractal Júlia, bem como sua lei de geração. Vale ressaltar que os fractais se encontram em diversas aplicações da vida real, além de ser muito utilizado como ferramenta da natureza. Por curiosidade, um fractal muito conhecido que pode ser observado na natureza é o romanesco, onde sua estrutura de repetição permite uma eficiência muito alta de cobertura: Área e perímetro. Cujos elementos são necessários para a sobrevivência e absorção de nutrientes e iluminação para a planta em questão.

2 Estratégia de paralelização

Inicialmente o resultado foi satisfatório, tendo em vista a uma atividade do paralelismo do algoritmo. Tomando como exemplo os resultados de tempo de execução, considerando uma imagem Quadrada nxn sendo n = 10.000, speed-up médio foi aproximadamente 2.45. Vale ressaltar que o desenvolvimento deixa o algoritmo em Obmep não fiz uso de offload, isto é, a execução foi inteiramente feita no processador.

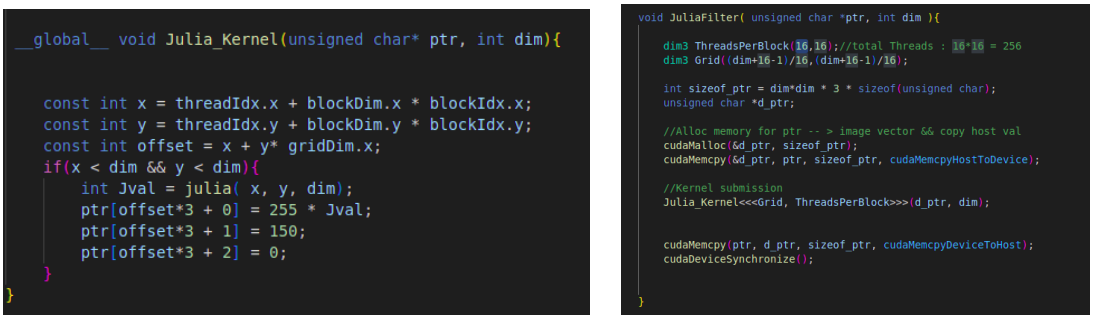
```
#ifdef OPENMP
void JuliaFilter( unsigned char *ptr, int DIM ){
    int y,x,offset;
    #pragma omp parallel for private(y, x, offset) shared(ptr)
    for (y=0; y<DIM; y++) {
        for (x=0; x<DIM; x++) {
            offset = x + y * DIM;

            int Jval = julia( x, y, DIM);
            ptr[offset*3 + 0] = 255 * Jval;
            ptr[offset*3 + 1] = 150;
            ptr[offset*3 + 2] = 0;
        }
    }
}
#endif
```

Figura 1: Algoritmo em openMP

Na imagem acima1, é possível notar principais diferenças em relação o código se-

rial: Utilização de pragma e classificação do tipo de variável. Este método de programação, conhecido pelo uso de diretivas de programação, permite ao compilador o melhor manejo de threads, criando execuções paralelas independentes em cada núcleo de processamento. Voltando ao algoritmo, é declarado como privado as variáveis x y e offset, seguida da declaração de compartilhada: PTR. O motivo desta classificação se deve por independência de dados e compartilhamento de memória, respectivamente.
Vale ressaltar que o formato da imagem é pnm, que permite maior facilidade na manipulação da imagem - a compressão de imagem não será abordada neste projeto, pois foge do escopo em questão.

A facilidade de manipulação de dados para transmissao de dados abstrai e facilita o desenvolvimento inicial de programas em paralelo. Isso permite com que o programador se preocupe apenas com a classificação dos tipos de variáveis, diferente de programações do tipo SYCL e CUDA. Estas linguagens permitem a a liberdade com o programador em relação à transmissão e controle dos dados, oque dificulta seu estilo de programação. Vale ressaltar que as duas linguagens mencionadas operam em aceleradores sendo CUDA especificamente para placas de vídeo da NVidia, e SYCL oferece maior heterogeneidade na sua programação: Fpga, GPU e CPU.



```
__global__ void Julia_Kernel(unsigned char* ptr, int dim){

    const int x = threadIdx.x + blockDim.x * blockIdx.x;
    const int y = threadIdx.y + blockDim.y * blockIdx.y;
    const int offset = x + y* gridDim.x;
    if(x < dim && y < dim){
        int Jval = julia( x, y, dim);
        ptr[offset*3 + 0] = 255 * Jval;
        ptr[offset*3 + 1] = 150;
        ptr[offset*3 + 2] = 0;
    }
}
```

(a)

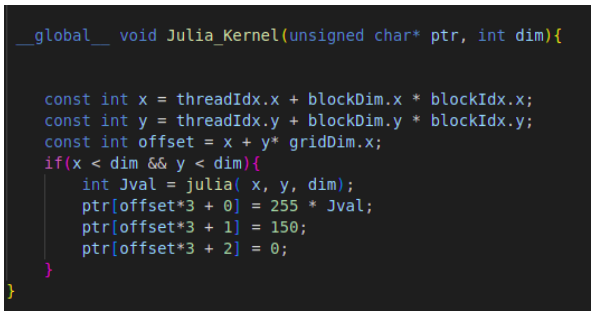
```
void JuliaFilter( unsigned char *ptr, int dim ){

    dim3 ThreadsPerBlock(16,16);//total Threads : 16*16 = 256
    dim3 Grid((dim+16-1)/16,(dim+16-1)/16);

    int sizeof_ptr = dim*dim * 3 * sizeof(unsigned char);
    unsigned char *d_ptr;

    //Alloc memory for ptr -- > image vector && copy host val
    cudaMalloc(&d_ptr, sizeof_ptr);
    cudaMemcpy(&d_ptr, ptr, sizeof_ptr, cudaMemcpyHostToDevice);

    //Kernel submission
    Julia_Kernel<<<Grid, ThreadsPerBlock>>>(d_ptr, dim);


    cudaMemcpy(ptr, d_ptr, sizeof_ptr, cudaMemcpyDeviceToHost);
    cudaDeviceSynchronize();


}
```

(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.

O primeiro algoritmo apresenta o encapsulamento das chamadas e transmissão de



```
__global__ void Julia_Kernel(unsigned char* ptr, int dim){


    const int x = threadIdx.x + blockDim.x * blockIdx.x;
    const int y = threadIdx.y + blockDim.y * blockIdx.y;
    const int offset = x + y* gridDim.x;
    if(x < dim && y < dim){
        int Jval = julia( x, y, dim);
        ptr[offset*3 + 0] = 255 * Jval;
        ptr[offset*3 + 1] = 150;
        ptr[offset*3 + 2] = 0;
    }
}
```

(a) (b)

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

dados para o GPU, abstraindo questões específicas da configuração da placa de vídeo. Um intuito deste encapsulamento é facilitar o entendimento do código. É possível notar por exemplo o total de threads a serem disparados, bem como o total de blocos necessários para o desenvolvimento deste algoritmo. De modo a facilitar a explicação, será declarado a seguir as etapas de execução presentes no encapsulamento:

- declaração do total de blocos e threads
- alocação de memória no dispositivo
- disparo de threads
- cópia dos resultados para memória local(host)

Já na segunda imagem, é especificado o Kernel: Instrução a ser executada pela GPU. Nesta etapa, tem a declaração em X e Y- que é como será percorrida a imagem, por meio de um plano cartesiano - de modo que o conjunto imagem é escrito em RGB. Além disso existe a função declarada somente para o device, nomeada a Júlia. Essa função tem por objetivo verificar cada par(x,y) pertencente ao conjunto declarado. Isso é feito por meio de um loop de verificação, do qual verifica divergência por meio de uma condição$if(a.mag()) > 1000$. Essa condicional tem por objetivo separar pontos que pertencem ao conjunto.

```
int julia( int x, int y, int DIM) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    Complex c(-0.8, 0.156);
    Complex a(jx, jy);

    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;

        //Verifica Divergencia
        if (a.mag() > 1000)
            return 0;
    }
    return 1;
}
```

Figura 4: Condição de Existencia para Conjunto

3 Análise da escalabilidade: esperada e obtida

No que se refere à escalabilidade, esse código satisfaz este ponto. Isso isso pode ser notado por meio da implementação. Como pode-se observar, foi obtido um rendimento excelente de aceleração, seja GPU, seja CPU. Isso se deve ao fato deste algoritmo possuir independência entre resultados calculados, cujo comportamento é o ideal para execução em GPU. O paralelismo Expresso em placa de vídeo demonstra uma forte característica de granularidade Fina que se refere ao paralelismo. É possível notar também que o speed-up é proporcional o tamanho da imagem, isto é, quanto maior a dimensão da imagem, maior o speed-up relação ao serial.

4 Conclusão

Por fim há de se concluir, e consequente redução dos dados em posições menores corroboraram para um desempenho satisfatório. Possíveis melhorias podem ser feitas nesses códigos, das quais permitem agregar ainda mais o desempenho para o algoritmo, tal como alocação de memória local, acesso em banco de memória- para o caso das GPU's.

Referências Bibliográficas

[1] Conjunto julia(julia set). https://pt.wikipedia.org/wiki/Conjunto_de_Julia/. Accessed: 2023-04-8.