Universidade Federal de Alagoas – UFAL
Centro de Tecnologia – CTEC
Programa de Pós-Graduação em Engenharia Civil – PPGEC

Disciplina: Técnicas Computacionais Avançadas (EES108)
Professor: William Wagner Matos Lira
Período Letivo: 2022.1

Lista: 04
Data de Recolhimento: 07/08/2022

Discente: Gilberto Lucas Leandro dos Santos (2021106337)
Discente: Otávio Bruno de Araújo Rodrigues (2021105957)

Maceió, agosto de 2022

1. Introdução

O objetivo deste trabalho é implementar uma função que determine o menor pseudo-ângulo de um conjunto de pontos em relação a um ponto qualquer fornecido usando a linguagem OpenCL. Nas seções 2 e 3 são apresentadas o cálculo de pseudo-ângulo, bem como a linguagem para computação de alto desempenho. A função é implementada em *Python*, discutindo-se a performance e a eficiência da paralelização em função da quantidade de pontos.

2. Pseudo-ângulo

Em alguns problemas de geometria computacional é necessário avaliar funções trigonométricas (seno, cosseno, arco cosseno, etc). Bibliotecas disponíveis calculam essas funções, porém elas possuem um custo computacional elevado. Nesse sentido, para problemas de tamanho grande, perdas de desempenho significativas são observadas para alcançar sua solução. Existem problemas da geometria computacional que não é necessário o valor exato das funções trigonométricas, por exemplo: ordenação polar, onde apenas a comparação entre ângulos é realizada. Nesses casos, o uso de pseudo-ângulos é recomendável.

Segundo Lira (2002), a alternativa com maior apelo geométrico é substituir o círculo unitário por qualquer outra curva contínua que satisfaça a propriedade de que cada semi-reta partindo da origem a intercepta em um único ponto. Além disso, a medida do arco tomado sobre essa curva será uma função monótona do arco tomado sobre o círculo unitário. Para este trabalho, é adotado o quadro unitário (ver Figura 1) para o cálculo do pseudo-ângulo. Nele, o ângulo é substituído pelo comprimento do percurso nas arestas do quadrado partindo do ponto (1,0).

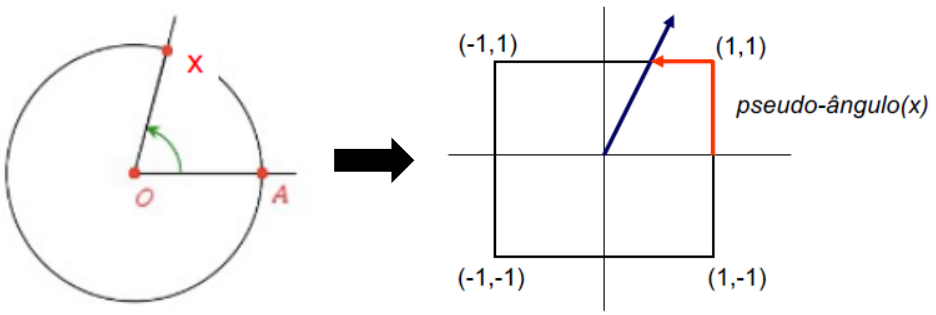

Figura 1 – Substituição do círculo unitário para o quadrado unitário.

Dessa forma, o cálculo do pseudo-ângulo pode ser realizado de modo a envolver apenas três comparações, uma soma e uma divisão (LIRA, 2002). É verificado qual quadrante o ponto pertence. Posteriormente, verifica-se se a semi-reta que parte do centro do quadrado até o ponto a ser avaliado intercepta a aresta do quadrado antes ou depois do vértice. Com isso é determinada a parte inteira do pseudo-ângulo, por semelhança de triângulos define-se sua fração.

3. Implementações

Nesta seção são apresentados os códigos desenvolvidos em *Python* para obter o cálculo do pseudo-ângulo.

3.1. Sequencial

O Algoritmo 1 apresenta a função que calcula o pseudo-ângulo. Essa função é chamada em uma estrutura de repetição para determinar o menor pseudo-ângulo de um conjunto de **n** pontos, conforme o Algoritmo 2.

```
def pseudo_angulo(X):
    n = len(X)
    if(n!=2):
        print("Vetor deve ser 2D\n");exit()
    else:
        x = X[0]
        y = X[1]
        if(y>0):
            if(x>0):                #Caso 1(Primeiro Quadrante)
                if(x>=y):
                    return y/x              #Caso 1.1
                else:
                    return 2 - x/y          #Caso 1.2
            elif(x==0):
                return 2           #Caso 1.3(Aresta Vertical Superior)
            else:                  #Caso 2(Segundo Quadrante)
                if(-x<=y):
                    return 2 + (-x)/y       #Caso 2.1
                else:
                    return 4 - y/(-x)       #Caso 2.2
        elif(y==0):
            if(x>0):
                return 8          #Caso 4.3 (Aresta Horizontal Direita)
            elif(x==0):
                return None       #Vetor Nulo
            else:
                return 4          #Caso 2.3(Aresta Horizontal Esquerda)
        else:
            if(x<0):              #Caso 3(Terceiro Quadrante)
                if((-x)>=(-y)):
                    return 4 + (-y)/(-x)    #Caso 3.1
                else:
                    return 6 - (-x)/(-y)    #Caso 3.2
            elif(x==0):
                return 6          #Caso 3.3(Aresta Vertical Inferior)
            else:                 #Caso 4(Quarto Quadrante)
                if(x<=(-y)):
                    return 6 + x/(-y)       #Caso 4.1
                else:
                    return 8 - (-y)/x       #Caso 4.2
```

Algoritmo 1 – Função em *Python* para determinação do pseudo-ângulo.

```
pseudo_a = []                    #vetor para armazenar os pseudo-angulos
t0_sequencial=time.time()         #tempo inicial
for i in range(n[i]):
  pseudo_a.append(pseudo_angulo([x_np[i],y_np[i]]))

menor_seq = min(pseudo_a)         #menor valor da lista

tf_sequencial = time.time()       #tempo final

#armazena o menor pseudo angulo para n pontos
menor_sequencial.append(menor_seq)

#armazena o tempo da solucao para n pontos
time_sequencial.append(tf_sequencial - t0_sequencial)
```

Algoritmo 2 – Estrutura de repetição para cálculo do menor pseudo-ângulo de n pontos.

3.2. Paralela com o OpenCL

Por meio da biblioteca *pyopencl* é possível paralelizar a rotina apresentada na seção anterior. Tem-se a geração de um contexto, onde é determinada qual placa gráfica será utilizada para solução do problema. O conjunto de pontos é copiado da aplicação em *Python* para o *buffer* global OpenCL para comunicação entre a aplicação e o *kernel*. Posteriormente, tem-se o cálculo do pseudo-ângulo de forma paralela e atribuição dos resultados. A determinação do menor pseudo-ângulo é realizada através de uma função própria do *Python*. Essas considerações podem ser observadas no Algoritmo 3. A função apresentada no Algoritmo 1 é adaptada para construir o *kernel* da linguagem OpenCL, como pode ser visto no Algoritmo 4.

```
#criacao do contexto
ctx1 = cl.create_some_context();queue1 = cl.CommandQueue(ctx1) ; mf1 = cl.mem_flags
prg1 = cl.Program(ctx1, """...  #Ver algoritmo 4

t0_paralelo_1=time.time()
#Copia para o buffer
x_g = cl.Buffer(ctx1, mf1.READ_ONLY | mf1.COPY_HOST_PTR, hostbuf=x_np)
y_g = cl.Buffer(ctx1, mf1.READ_ONLY | mf1.COPY_HOST_PTR, hostbuf=y_np)
res_g = cl.Buffer(ctx1, mf1.WRITE_ONLY, x_np.nbytes)
#chama a funcao do kernel
prg1.pseudo(queue1, x_np.shape, None, x_g,y_g, res_g)
#inicializa o vetor com o resultado (valores aleatorios)
res_np = np.empty_like(x_np)
#atribui os valores da soma na variavel de resultado
cl.enqueue_copy(queue1, res_np, res_g)
#menor pseudo angulo
menor_paralelo_1.append(min(res_np))
tf_paralelo_1=time.time()
time_paralelo_1.append(tf_paralelo_1 - t0_paralelo_1)
```

Algoritmo 3 – Solução em paralelo do menor pseudo-ângulo de n pontos.

```
prg1 = cl.Program(ctx1, """__kernel void pseudo(__global const float *x_g,
__global const float *y_g, __global float *res_g){
  int gid = get_global_id(0);
  if(y_g[gid] > 0){
    if(x_g[gid] > 0){                          //Caso 1  (Primeiro Quadrante)
      if( x_g[gid] >= y_g[gid]){
        res_g[gid] = y_g[gid]/x_g[gid];        //Caso 1.1
      }else{
        res_g[gid] = 2 - x_g[gid]/y_g[gid];}   //Caso 1.2
    }else if (x_g[gid] == 0){
      res_g[gid] = 2;                      //Caso 1.3  (Aresta Vertical Superior)
    }else{                                     //Caso 2 (Segundo Quadrante)
    if(-x_g[gid]<=y_g[gid]){
      res_g[gid] = 2 + (-x_g[gid]/y_g[gid]);   //Caso 2.1
    }else{
      res_g[gid] = 4 - (y_g[gid]/(-x_g[gid]));}} //Caso 2.2
  }else if (y_g[gid]==0){
    if(x_g[gid]>0){
      res_g[gid] = 8;                   //Caso 4.3  (Aresta Horizontal Direita)
    }else if (x_g[gid] == 0){
      printf("%s",  "Vetor Nulo");
      res_g[gid] = NULL;                       //Vetor Nulo
    }else{
      res_g[gid] = 4;}                 //Caso 2.3  (Aresta Horizontal Esquerda)
  }else{
    if(x_g[gid]<0){                     //Caso 3    (Terceiro Quadrante)
      if((-x_g[gid])>=(-y_g[gid])){
        res_g[gid] = 4 + (-y_g[gid])/(-x_g[gid]);          //Caso 3.1
        }else{
        res_g[gid] = 6 - (-x_g[gid])/(-y_g[gid]);}         //Caso 3.2
    }else if(x_g[gid]==0){
      res_g[gid] = 6;                     //Caso 3.3  (Aresta Vertical Inferior)
    }else{
    if(x_g[gid]<=(-y_g[gid])){         //Caso 4    (Quarto Quadrante)
      res_g[gid] = 6 + x_g[gid]/(-y_g[gid]);         //Caso4.1
    }else{
      res_g[gid] = 8 - (-y_g[gid])/x_g[gid];}}         //Caso4.2
  }
}
""").build()
```

Algoritmo 4 – Criação do contexto e *Kernel* em Open CL para determinação do pseudo-ângulo.

4. Resultados

Nesta seção são apresentados os resultados de tempo computacional para a verificação do menor pseudo-ângulo em conjuntos com 10000, 100000, 1000000, 10000000, 100000000 valores. Tais análises foram executadas em um computador pessoal com processador Intel Core i5-10300U CPU 2.50 GHz, com 4 núcleos, 8 processadores lógicos e memória RAM de 8 GB**.** Foi utilizada uma GPU do tipo NVIDIA GTX 1650 GDDR6 com 4 GB. Mais informações sobre esta são apresentadas na Figura 2.

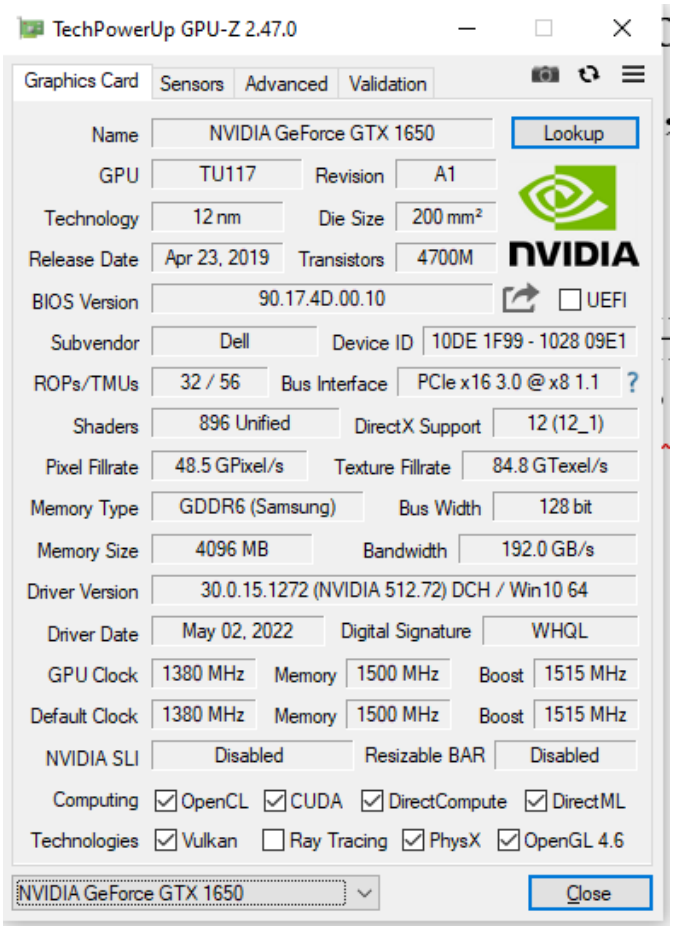

Figura 2 – Configurações da GPU NVIDIA GTX 1650 GDDR6.

A Figura 3 apresenta um gráfico com o tempo computacional pela quantidade de valores do conjunto, conforme as implementações sequencial e paralela. Como esperado, a implementação sequencial resultou em tempos maiores para todos os casos. A Tabela 1 detalha a superioridade apresentada no gráfico, onde no caso mais crítico o algoritmo sequencial chega a ser 88 vezes mais rápido para uma amostra com 100000 elementos. Nota-se que após esse pico de diferença, o algoritmo paralelo mantém-se numa faixa 60 vezes maior.

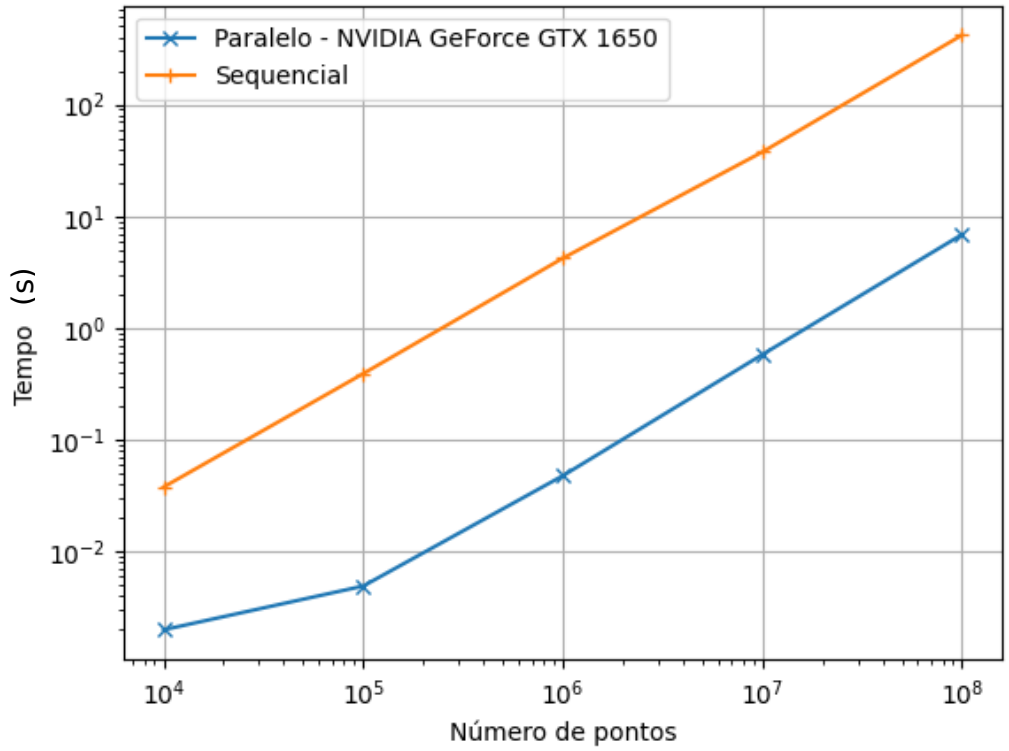

Figura 3 – Comparação do tempo computacional entre as implementações sequencial e paralela.

Tamanho da amostra	Tempo sequencial (s)	Tempo paralelo (s)	Sequencial/Paralelo
10000	0,037926	0,002007	19
100000	0,391813	0,004938	79
100000	4,218076	0,047872	88
1000000	37,60353	0,580699	65
10000000	416,9749	6,876208	61

Tabela 1 - Tempo computacional para os códigos sequencial e paralelo para grandes amostras.

É importante destacar que em amostras menores a diferença de tempo entre as implementações sequencial e paralela é desprezível, chegando até a um desempenho melhor sem o OpenCL, conforme ilustrado na Tabela 2. O tempo nulo da implementação sequencial indica um tempo menor que a precisão de 1e-6 s definida. Além disso, o tempo da implementação paralela é praticamente constante, o que pode estar relacionado com algum tempo de compilação ao utilizar a GPU.

Tamanho da amostra	Tempo sequencial (s)	Tempo paralelo (s)	Sequencial/Paralelo
10	0	0,000997	0
100	0	0,000997	0
1000	0,002992	0,000996	3

Tabela 2 - Tempo computacional para os códigos sequencial e paralelo para pequenas amostras.

5. Conclusões

Com base nos resultados obtidos, destaca-se a eficiência de implementações paralelas, reduzindo significativamente o tempo em relação ao código sequencial na determinação do pseudo-ângulo a partir de um conjunto de pontos, principalmente, quão maior for a quantidade de valores. No caso de amostras pequenas (<10000), o tempo paralelo chega a ser constante, sendo suficiente a implementação sequencial.

6. Referências

Lira, W. W. M., Técnicas Computacionais Avançadas, Notas de Aula. Universidade Federal de Alagoas – UFAL, 2022.