Conversão de algoritmos de processamento e análise de dados atmosféricos da linguagem C para CUDA

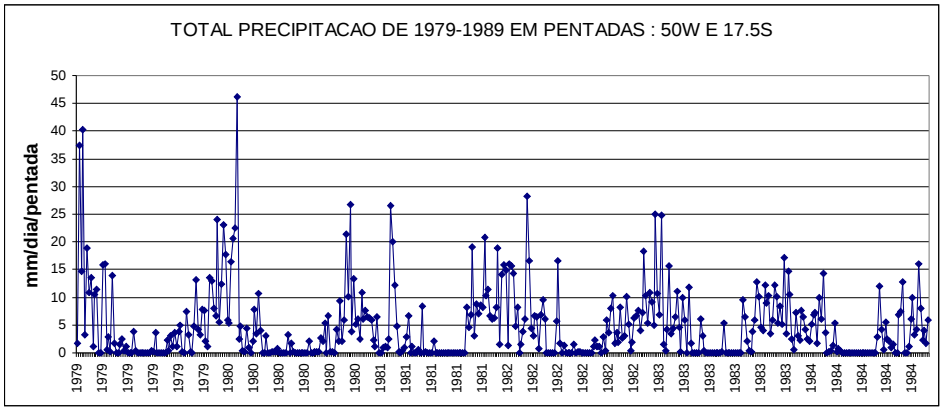
Bruno Nocera Zanette - *brunonzanette@gmail.com*
Prof. Fabiano Silva - *fabiano@inf.ufpr.br*

- Uma análise do custo e benefício do processo de conversão de um algoritmo de processamento e análise de dados atmosféricos de C para CUDA

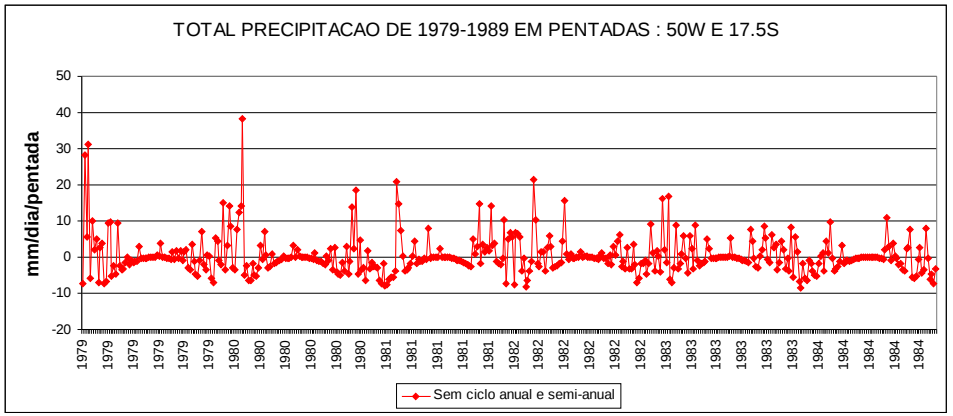
- Preocupação com a utilização do código por pesquisadores e estudantes de iniciação científica, e não somente com a performance

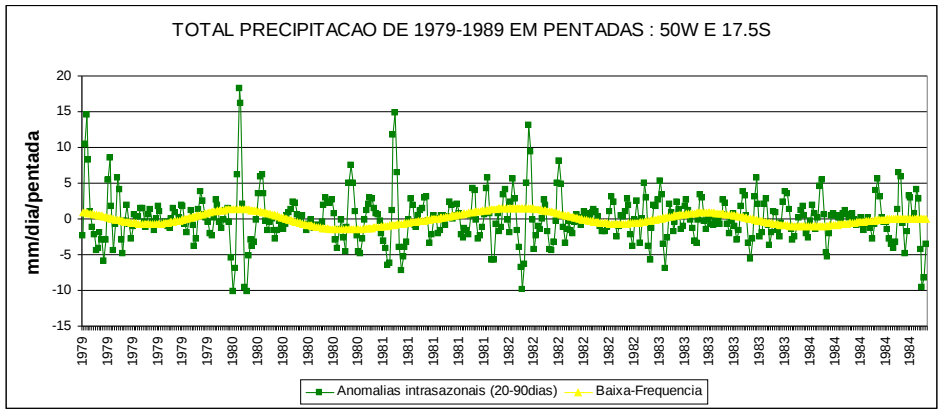
O algoritmo “Filtro de Lanczos”

- A filtragem de uma série de dados permite reter nela apenas variações temporais com períodos (ou frequências) que nos interessam estudar. O filtro de Lanczos é um dos métodos que permite fazer isto.



$$Res[T+\lfloor(K/2)\rfloor]=\sum_{i=0}^{K-1} WT[i]*Dados[T+i]$$



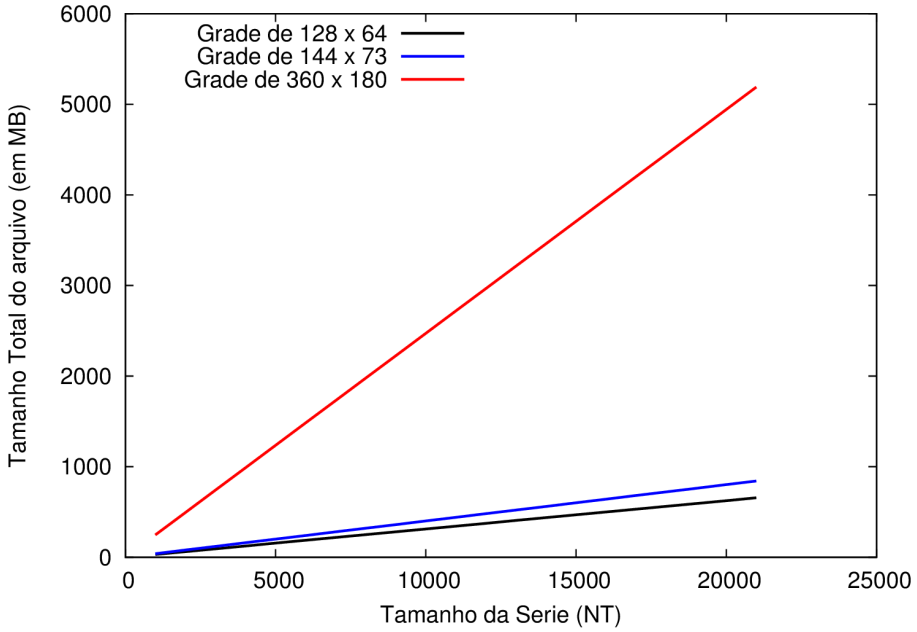


Algoritmo e explicação cedida por:
Alice M. Grimm – Dept. Física da UFPR

Pontos fortes e fracos

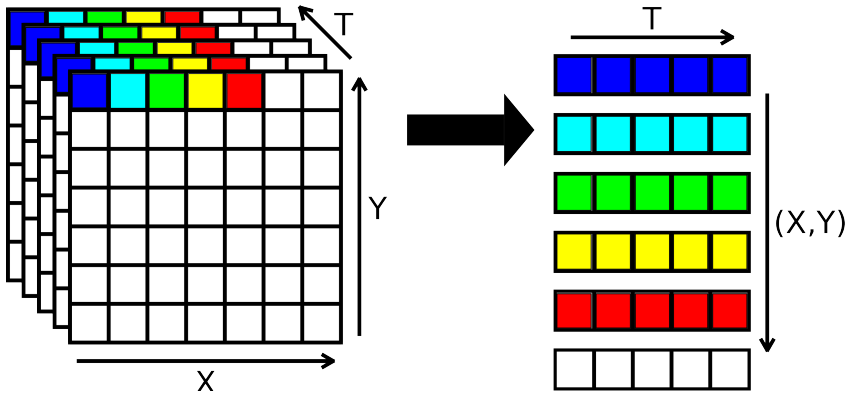
- Cada quadrícula é processada de forma independente
- Muitas contas para o cálculo de um único valor. E para cada valor de entrada, um de saída
- Dados de entrada grandes e memória da GPU pequena
  - Grades de 144x73 ou 360x180 quadrículas, por exemplo...
  - São utilizados dados diários. Logo, para apenas 10 anos de dados são 3650 valores por quadrícula

1° = ~111Km



Solução

- Processar cada quadrícula paralelamente
- Usar ciclos de processamento, em que cada ciclo seja processado apenas uma fração das quadrículas, de forma que o tamanho dos dados usados em cada ciclo (entrada e saída) possam ser armazenados na GPU
- Porém, os dados são originalmente organizados de forma que os valores dos eixos X e Y fiquem juntos, fazendo com que os valores de uma quadrícula no eixo do tempo fiquem separados
- Para isso, foi utilizada uma função de leitura que armazena os dados na memória do Host de forma que a série de tempo de cada quadrícula seja contínua, assim sendo possível copiar pedaços separados desse dado para a GPU



Implementação

```
int main(int argc, char **argv){

<DECLARACAO_DAS_VARIAVEIS>
<ALOCA OS DADOS DE ENTRADA E SAIDA NO HOST>
<LE OS ARGUMENTOS E OS DADOS DE ENTRADA>
<CALCULA O VETOR "WT">
<CALCULA OS PARAMETROS DE EXECUCAO DO CUDA>

//ALOCA OS DADOS DE ENTRADA E SAIDA NO DEVICE
cudaMalloc((void**)&d_entrada, tam_por_ciclo);
cudaMalloc((void**)&d_saida, tam_por_ciclo);

//ALOCA E COPIA O VETOR "WT" DO HOST PARA O DEVICE
cudaMalloc((void**)&d_wt, (NWT*sizeof(float)));
cudaMemcpy(d_wt, h_wt, (NWT*sizeof(float)),
 cudaMemcpyHostToDevice);
```

- Executado nas duas versões
- Exclusivo da versão CUDA
- Exclusivo da versão Sequencial

Implementação

```
for (ciclo=0;ciclo<total_ciclos;ciclo++)
    pos_entrada=(ciclo*npos_por_ciclo)*NT;
    cudaMemcpy(d_entrada,(h_entrada+pos_entrada),
         tam_por_ciclo,cudaMemcpyHostToDevice);
    filtro_lanczos <<< … >>> ( … );
    pos_saida=(ciclo*npos_por_ciclo)*NT;
    cudaMemcpy((h_saida+pos_saida),d_saida,
         tam_por_ciclo,cudaMemcpyDeviceToHost);
}

salva_arq_saida(param, h_saida);
desaloca_variaveis();
return 0;
}
```

- Executado nas duas versões
- Exclusivo da versão CUDA
- Exclusivo da versão Sequencial

Implementação

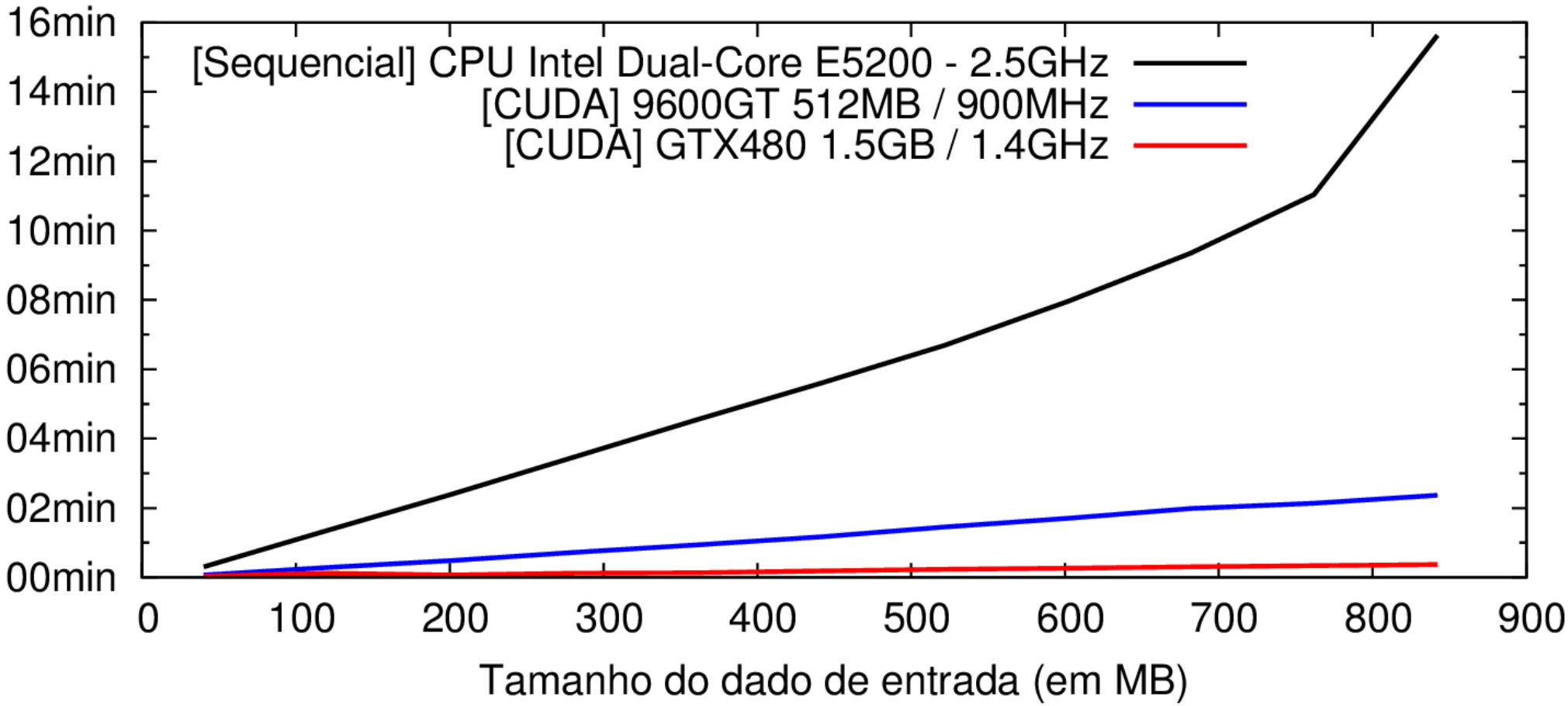
```
__global__ void filtro_lanczos( <PARAMETROS > ){

    <DECLARACAO_DE_VARIAVEIS>

    int p=(blockDim.x*blockIdx.x)+threadIdx.x;
    if (p >= total_pos) return;

    for (p=0;p<total_pos;p++){
        ini_seq=(p*NT);
        for (j=fNWL;j<NT-fNWL;j++){
            <CALCULOS>
            s[ini_seq+j]=resultado;
        }
    }
}
```

- Executado nas duas versões
- Exclusivo da versão CUDA
- Exclusivo da versão Sequencial

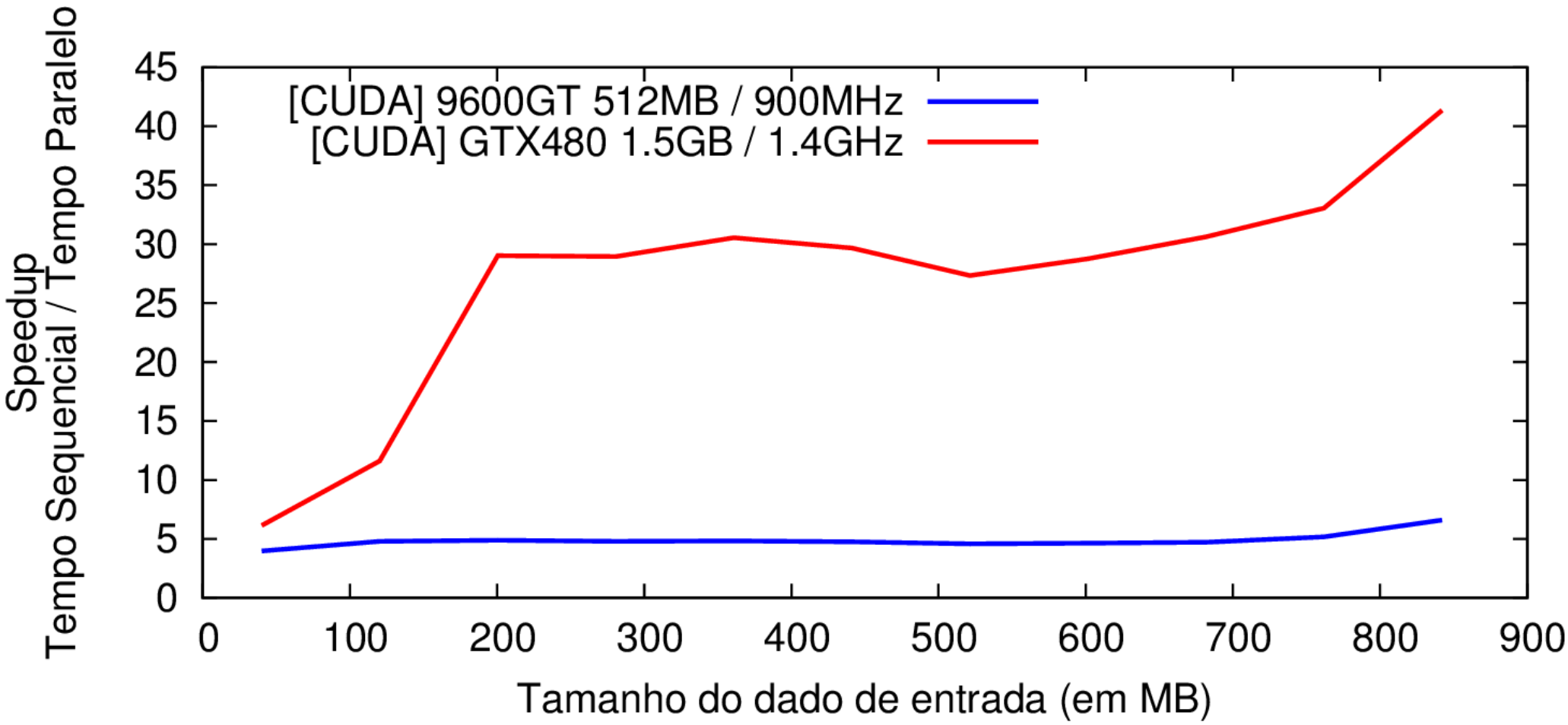
Resultados

- Para comparar o desempenho da implementação em CUDA, o código sequencial e em CUDA foram executados exatamente com o mesmo dado de entrada e opções do filtro de Lanczos, variando apenas o tamanho da série de tempo
- Foi utilizado como parâmetro de execução 8 threads por bloco. O número total de blocos e ciclos foram calculados diretamente pelo programa em relação ao tamanho do dado de entrada
- Não foi feito nenhum tipo de calibração mais otimizada. Tal decisão foi feita propositalmente, em vista que tais ajustes podem variar de uma máquina para outra, ou até mesmo de uma entrada para outra, e assumindo que tais programas deverão ser executados por pesquisadores e que os mesmos não teriam tempo e/ou conhecimento para fazer tais ajustes.

Resultados


16min
14min
12min
10min
08min
06min
04min
02min
00min
[Sequencial] CPU Intel Dual-Core E5200 - 2.5GHz
[CUDA] 9600GT 512MB / 900MHz
[CUDA] GTX480 1.5GB / 1.4GHz
0
100
200
300
400
500
600
700
800
900
Tamanho do dado de entrada (em MB)

Resultados


Speedup
Tempo Sequencial / Tempo Paralelo
[CUDA] 9600GT 512MB / 900MHz
[CUDA] GTX480 1.5GB / 1.4GHz
45
40
35
30
25
20
15
10
5
0
0
100
200
300
400
500
600
700
800
900
Tamanho do dado de entrada (em MB)

Conclusões e Propostas

- Nesse trabalho foi proposto uma implementação simplificada de um algoritmo de processamento de dados para CUDA, e mesmo sem muitas otimizações foi obtido resultados muito bons!
- Além disso, a conversão do código C em CUDA se mostrou simples o bastante para que pesquisadores e estudantes de iniciação científica possam utilizar e alterar.
- **Possibilidade de utilização do mesmo método e estruturas de dados para a conversão de outros algoritmos!**
- Porém, o método não é aplicável (ou necessita de muito mais esforço) a algoritmos mais complexos.