Bruno Nocera Zanette

Análise do custo-benefício da conversão de algoritmos de processamento de dados meteorológicos de C para CUDA

Curitiba, Janeiro de 2013

Bruno Nocera Zanette

Análise do custo-benefício da conversão de algoritmos de processamento de dados meteorológicos de C para CUDA

Trabalho de conclusão de curso de Bacharelado em Ciência da Computação.

Orientador:
Prof. Fabiano Silva

UNIVERSIDADE FEDERAL DO PARANÁ

Curitiba, Janeiro de 2013

Lista de Abreviaturas e Siglas

Host - Máquina que hospeda a placa-de-vídeo

Device - Placa-de-vídeo capacitada com a tecnologia CUDA

NOAA - National Oceanic & Atmospheric Administration

Lista de Figuras

Sumário

Resumo

Neste trabalho farei um estudo de caso do processo de conversão de um algoritmo de processamento de dados da linguagem C para CUDA, com o intuito de mostrar que a relação custo e benefício dessa tarefa pode ser muito boa. Além disso o trabalho tem como um segundo objetivo criar um modelo de código para que este possa ser usado como suporte para a conversão de novos algoritmos.

1 Introdução

A meteorologia é uma das áreas que mais exige poder computacional para alcançar seus objetivos. A previsão meteorológica é o exemplo mais claro disso. Tal tarefa necessita de supercomputadores, como por exemplo o novo supercomputador Cray XT-6 adquirido pelo CPTEC/INPE considerado um dos 50 mais rápidos do mundo, e equipes especializadas em otimizar o código-fonte dos programas para ser capaz de alcançar bons resultados. Isso acontece porque para alcançar tamanha precisão é necessário que os dados utilizados como base tenham uma resolução igualmente boa. Porém isso faz com que a quantidade de dados a ser processado seja muito maior, e assim elevando o tempo de execução.

Entretanto, o campo da meteorologia não se resume apenas a previsão atmosférica. Existem muitos laboratórios espalhados pelo mundo que realizam pesquisas igualmente importantes e que enfrentam problemas similares aos já citados. A diferença é que por um lado tais pesquisas não necessitam de dados com tanta precisão, mas que por outro não possuem tanto investimento para equipamentos melhores. Outra grande diferença é que nessas pesquisas são os próprios pesquisadores, normalmente da área de física ou engenharia, que escrevem e executam os programas. Uma consequência disso é que os programas normalmente não são escritos da forma mais eficaz, fazendo com que algoritmos simples tenham um tempo elevado de execução, e assim desperdiçando o hardware disponível.

Nesse trabalho será estudado métodos para otimizar algoritmos de processamento e análise de dados atmosféricos com base em técnicas de programação paralela, visando auxiliar programadores com pouco experiência a reduzir o tempo de execução de algoritmos. Nos próximos capítulos será detalhado a organização dos dados utilizados, as técnicas utilizadas e os resultados obtidos.

2 *Algoritmos atmosféricos*

O objetivo desses algoritmos é processar uma base de dados, com base em conceitos estatísticos, de tal forma a gerar novas informações que possam ser usadas em pesquisas e estudos relacionada a área atmosférica. Grande parte desses algoritmos se baseiam no processamento de uma única série de dados e calculam a informação para apenas essa série, da mesma forma como é feito quando calcula-se a média de uma série de valores. Na verdade, muito desses algoritmos também podem serem usados em outras áreas com a diferença do tipo do dado de entrada usado e alguns pequenos ajustes. Posteriormente esses algoritmos são aplicados para um conjunto de séries relacionadas a diferentes locais da área a ser estudada, como será descrito nos próximos capítulos. Um fato importante a se notar é de que, normalmente, o processamento de cada uma das séries é independente uma das outras.

Para exemplificar usaremos os algoritmos Filtro de Lanczos e Teste de Monte Carlo, pois cada um possui caraterísticas próprias que fazem com que a execução dos mesmos possa ser muito demorada quando executados sequencialmente.

2.1 Filtro de Lanczos

O algoritmo Filtro de Lanczos, descrito por (DUCHON, 1979), é utilizado, entre outras coisas, para filtrar variações temporais de dados atmosféricos diários. O problema consiste em calcular resultados para um certo vetor de dados, multiplicando-se um vetor de constantes de tamanho menor, WT, previamente calculado, com um pedaço da serie de dados, e armazenando o resultado numa nova série de dados, na posição central, como descrito na fórmula a seguir:

$$Res[T + \lfloor \frac{K}{2} \rfloor] = \sum_{i=0}^{k-1} WT[i] \times Dados[T + i]$$

Para cada resultado produzido por esse algoritmo é necessário K multiplicações, onde K é o número de pesos usados, definidos por (DUCHON, 1979). Logo, para um série de 100 valores

é necessário aproximadamente $K * 100$ operações. Uma vez que os dados de entrada usados para esses algoritmos possuem centenas de séries de centenas de valores cada uma, como será descrito mais a frente, o tempo de execução desse algoritmo, mesmo sendo um sequencial um função relativamente simples, tende a ser muito grande.

2.2 Teste de Monte Carlo

O teste de Monte Carlo é utilizado, dentro da área de estudos atmosféricos, para calcular a significância entre duas séries distintas. Esse algoritmo faz parte de uma classe de algoritmos que utilizam o método de Monte Carlo, o qual se baseia na observação de valores aleatórios e o uso dessa amostra para o cálculo da função de interesse. Outra aplicação muito comum desse método é o cálculo de integrais.

O algoritmo a ser estudado nesse trabalho é explicada, de acordo com (SABOIA, 2010), pelos seguintes passos, sendo n_e o número de experimentos do teste de Monte Carlo:

1. é calculado o coeficiente de correlação entre as séries A e B. Por simplicidade, a correlação entre essas duas séries será chamada de correlação original;
2. a séria A sofre permutação entre seus membros, a fim de se formar uma nova série;
3. é calculado o coeficiente de correlação entre esta nova série e a série B;
4. compara-se o novo coeficiente de correlação com o anterior;
5. repete-se os passos 2,3 e 4, n_e vezes;
6. chamando de cor_m o número de vezes em que o novo coeficiente de correlação foi maior do que o original, o nível de significância é definido como a razão entre cor_m e n_e;

A execução desse algoritmo pode ser bastante elevado dependendo do valor n_e, pois o mesmo define a quantidade de novas séries a serem geradas e o cálculo de um novos coeficiente de correlação. Sabendo-se que essas séries podem conter centenas de valores, a quantidade de acessos a memória necessário para escrever e ler todos esses valores é muito grande, elevando o tempo de execução consideravelmente.

3 *Organização dos dados*

Os dados utilizados como entrada para esses algoritmos são baseados em séries temporais de valores que representam o comportamento de alguma variável atmosférica, tal como quantidade de chuva e temperatura do ar, num local específico em vários momentos do tempo. Essas informações podem ser obtidas por satélites, balões atmosféricos, estações meteorológicas, entre outros, e normalmente são obtidas uma ou mais vezes por dia. A partir da obtenção dessas séries de dados em diversos locais espelhados por uma determinada área, ou até mesmo no globo terrestre inteiro, são feitas verificações para a validação desses valores obtidos e executado métodos para a dedução dos valores referentes a localidades onde não é possível fazer a medição, como por exemplo no meio dos oceanos. Enfim é criado uma grade que divide igualmente a área de interesse em pequenas quadrículas, de tal forma que cada uma dessas áreas seja representada por apenas uma série de dados, e de que a visualização como um todo dessas várias séries represente o comportamento da variável em questão na área total.

Fica claro que a utilização de apenas uma série de dados para cada uma dessas áreas pode gerar muitas informações equivocadas, uma vez que o comportamento dessas variáveis atmosféricas podem variar bruscamente dentro de cada uma dessas regiões. Por isso há a necessidade de que o tamanho dessas regiões seja o menor possível. Entretanto isso faz com que haja muito mais séries para a mesma área de interesse, o que numa escala global faz com que o número de quadrículas cresça bruscamente.

Existem diversas instituições que fornecem livremente esses dados ao público, como o (NOAA,), o (MET...,) e a (NASA,). São esses dados que servem como base para estudos sobre mudanças climáticas.

3.1 Estrutura de dados

A estrutura de dados usada para armazenar essas informações é uma matriz tri-dimensional (X,Y,T) na qual os eixos X e Y se referem a posição espacial e o eixo T ao tempo, e cada

um desses eixos possui uma quantidade fixx de valores, respectivamente NX, NY e NT. Essa estrutura é armazenada de forma sequencial variando primeiramente os eixos X e Y e por fim o eixo T, ou seja, os valores do eixo T para cada ponto (X,Y) ficam separados.

3.2 Tamanho dos dados

Um dos principais desafios enfrentados no uso desses dados é o tamanho dos arquivos necessários para armazenar toda a informação, que pode ser calculado multiplicando-se o número total de quadrículas (NX * NY) pelo tamanho de cada uma das séries de dados (NT). Por exemplo, para armazenar as informações referente a 1 ano de valores diários, para uma grade global de 360 por 180 quadrículas são necessários 94608000 bytes (NT=365 x [NX=180 * NY=360] x 4 bytes), ou aproximadamente 90MB. Pode não parecer muito mas, sabendo-se que cada uma dessas quadrículas abrange uma área de 111Km2 e são as informações de apenas 1 ano, percebe-se que para um perído de dados um pouco maior ou para quadrículas com uma área um pouco menor esse valor pode chegar facilmente nas casa dos GigaBytes. Prova disso é que, para esse mesmo exemplo citado, caso fosse um período de 50 anos seriam necessários 4.4GBs.

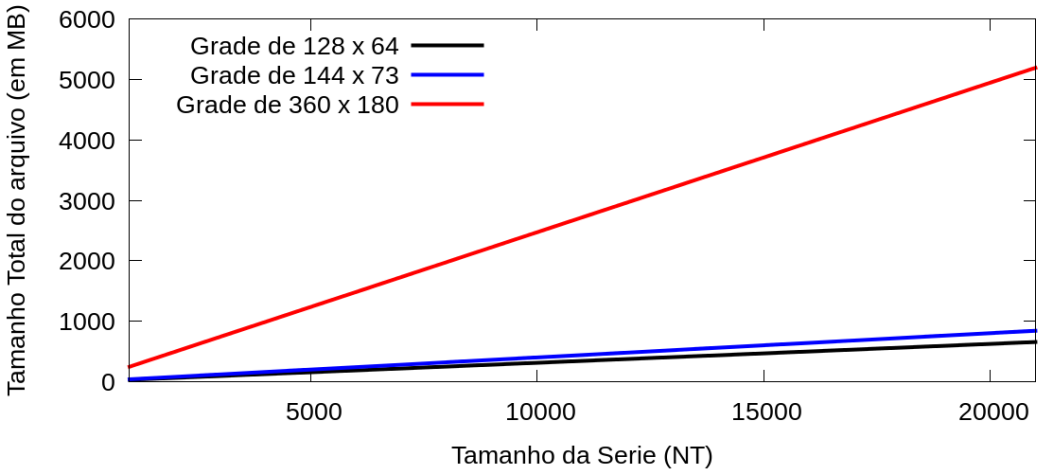

Figura 3.1: Gráfico do tamanho dos dados

4 *Implementação Paralela dos algoritmos*

Como descrito anteriormente, o processamento de cada quadrícula de dados é feita de forma independente uma da outra, fazendo com que a implementação paralela desses algoritmos seja algo natural a se fazer. Entretanto, existem alguns problemas a serem enfrentados para o sucesso dessa tarefa, especialmente ao usar a plataforma CUDA.

Nesse capítulo será analisado as principais etapas e dificuldades das implementações paralela em OpenMP e CUDA dos algoritmos Filtro de Lanczos e Teste de Monte Carlo.

4.1 Implementação em OpenMP

A maior vantagem em se usar as bibliotecas OpenMP para implementar soluções paralelas desses algoritmos é a facilidade. Isso se dá primeiramente pelo fato de que não é necessário um tratamento especial da memória, diferente da implementação em CUDA, e principalmente por se tratar da adição de pouquíssimas linhas de código com a função de apenas especificar como a função em questão deve ser paralelizada.

Esses fatores são especialmente importantes no caso dos algoritmos estudados pois o fato de não haver mudanças mais profundas no código há uma maior confiança nos resultados obtidos. Além disso, o código é capaz de rodar em qualquer processador e não é necessário a reconfiguração de nenhum parâmetro para que a execução em máquinas diferentes seja a melhor possível, pois a configuração do número de threads já é feita durante a execução. Isso faz com que seja muito fácil o uso dessa solução por parte dos pesquisadores em qualquer máquina.

Porém, como será descrito mais a frente, essa solução possui uma escalabilidade muito pequena, e em alguns casos o tempo de execução pode ser até maior do que a implementação sequencial.

4.2 Implementação em CUDA

Diferente da solução com base em OpenMP, a implementação em CUDA possui uma grande escalabilidade e reduziu significantemente o tempo de execução em todos os testes realizados. Porém, a sua implementação necessita de muito mais cuidados, além de uma placa-de-vídeo que suporte a tecnologia CUDA.

Outros fatores importantes a se notar é que essas placas possuem uma memória própria e que a cópia da memória entre o Host e o Device não é feita automaticamente a quantidade, além de que a quantidade dessa memória existente é limitada e não há a possibilidade de expansão. São esses fatores que fazem com que a implementação seja mais difícil, especialmente no caso dos algoritmos estudados os quais, como já explicado, necessitam de uma grande quantidade de memória.

4.2.1 Proposta de solução

Uma solução para possibilitar o uso desses algoritmos em qualquer plataforma CUDA, independente do tanto de memória disponível, é processar esses dados em ciclos de processamento, ou seja, processar apenas uma parte dos dados de entrada a cada ciclo. Porém os mesmos 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, o oposto do que seria o ideal. Isso não é nenhum obstáculo quando pensamos numa execução sequencial do algoritmo onde a quantidade e facilidade de acesso à memória é maior, mas impossibilita a ideia de ciclos de processamentos, pelo fato dos valores estarem separados no eixo do tempo, o que impede que os mesmos sejam repartidos.

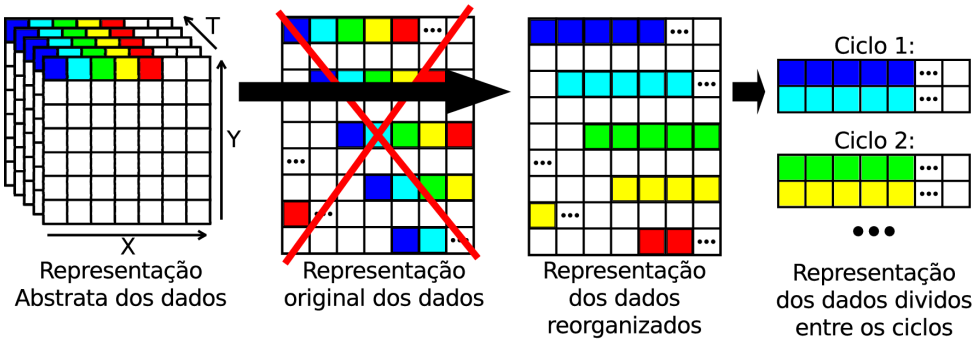

Figura 4.1: Organizacao dos dados

Para contornar isso foi utilizado uma função de leitura que armazena os dados na memória do Host de forma que a série de tempo de cada quadrícula seja contínua (4.1), assim sendo

possível copiar pedaços separados desse dado para a GPU. Além disso, ao invés da estrutura de matriz foi usado a estrutura de um vetor comum para armazenar esses valores e funções auxiliares para calcular a posição na memória em que se inicia os dados de cada quadrícula. Essa decisão ainda facilita a cópia de dados entre a memória do Host e do Device.

4.2.2 Implementação da solução

Em teoria essa tarefa seria a mais difícil do processo de implementação do algoritmo em CUDA, porém foi o que menos precisou de ajustes. Nas função que implementam os algoritmo em si as únicas alterações necessárias foram remover o loop de incremento das quadrículas a serem processadas, necessário na versão sequencial, e adicionar o cálculo que defini qual quadrícula cada um dos núcleos de processamento da GPU será responsável por processar.

Foi a função `Main` do código em C que sofreu as maiores mudanças. Porém, grande parte dessas alterações foram apenas para adicionar as premissas básicas de todo programa, como alocação e inicialização de memória, no contexto do CUDA. As outras modificações foram a adição das funções de cópia de memória entre Host e Device e um loop para controlar os ciclos.

Esses ciclos foram pensados para o caso da memória total necessária para armazenar os dados de entrada e saída for maior do que o total de memória disponível na GPU. Para que esse método funcione, além da organização dos dados previamente descrita, foi criado um pequeno algoritmo para dividir igualmente o processamento entre os vários ciclos. Dentro de cada um desses ciclos as seguinte ações são executadas, em ordem:

1. Cálculo da posição de início da cópia dos dados de entrada;
2. Cópia dos dados de entrada do Host para o Device;
3. Execução do algoritmo em questão;
4. Cálculo da posição de início da cópia dos dados de saída;
5. Cópia dos dados de saída do Device para o Host;

Todas essas ações possuem relações diretas com as ações consideradas praticamente obrigatórias em todas os programas escritos em C. São elas: leitura dos dados de entrada, execução do algoritmo de processamento, escrita dos dados de saída. Portanto, com exceção das alterações obrigatórias para adaptar o programa à plataforma CUDA, não foi feito nenhuma modificação ou otimização no código que implementa os algoritmos em si.

4.3 Comparação das alterações necessárias para a implementação em OpenMP e CUDA

```
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 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);

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);

    nome_algoritmo <<< … >>> ( … );

    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;
}
```

■ → Exclusivo da versão CUDA

Figura 4.2: Comparação da função `Main`

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

      <DECLARACAO_DE_VARIAVEIS>

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

      #pragma omp parallel private(p, ...)
      {
      for (p=0;p<total_pos;p++){
            <CALCULOS>
      }
      }
}
```

■ → Exclusivo da versão Sequencial e OpenMP
■ → Exclusivo da versão OpenMP
■ → Exclusivo da versão CUDA

Figura 4.3: Comparação da função que implementa o algoritmo

5 *Resultados*

Além da análise da facilidade de uso e implementação paralela desses algoritmos, para analisar o custo-benefício real das duas implementações é preciso primeiramente verificar se os resultados obtidos são corretos, e posteriormente o quão grande foi a redução do tempo de execução comparado a versão sequencial dos algoritmos. Nesse capítulo será estudado esses dois fatores.

5.1 Autenticação dos resultados

Algo imprescindível em algoritmos de processamento e análise de dados para que possam ser efetivamente usados em pesquisas é a confiabilidade dos resultados obtidos. Nem sempre é possível apenas observando os resultados saber se o mesmo esta certo ou errado. Portanto é preciso ter total confiança de que os dados obtidos da execução do algoritmo estão corretos.

Por esse motivo, o pequeno número de alterações feitas no código é tão importante, pois evita que erros sejam cometidos no processo de conversão. Além disso, em todos os testes realizados, os resultados obtidos da versão CUDA foram comparados com os do sequencial, e em todos os casos os mesmos foram idênticos, comprovando que os resultados estão corretos.

5.2 Avaliação do tempo de execução

Contudo, a principal motivação da implementação paralela desses algoritmos é a redução do tempo de execução e se tal melhora faz valer todo o esforço. Para comparar o desempenho das implementações em OpenMP e CUDA com a implementação sequencial foram realizados diversos testes, com diferentes parâmetros, e em diferentes configurações de Hardware, com o intuito de medir o tempo real de execução dos mesmos para então verificar se houve uma redução significativa desse tempo.

Nos testes da implementação em OpenMP foram utilizados os seguintes processadores:

Tabela 5.1: Modelos CPU

	AMD Opteron	**Intel I5**	**Intel I7**
Clock	2.8 GHz	3.3 Ghz	3.0 Ghz
Total de Threads	8	4	8

Já nos testes da implementação em CUDA foram utilizados as seguintes placas-de-vídeo:

Tabela 5.2: Modelos GPU

	NVidia 9600GT	**NVidia GTX480**
Clock	1.96 GHz	1.4 GHz
Total Memória	512 MB	1536 MB
Clock da Memória	900 Mhz	1848 Mhz
CUDA Cores	64	480
Threads por bloco	512	1024

Além disso, na versão CUDA foi utilizado como parâmetro de execução 64 threads por bloco. O número total de blocos e ciclos foram calculados diretamente pelo programa em relação ao tamanho do dado de entrada. Sendo assim, é perceptível que não foi feita nenhuma 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.

Os tempos foram marcados utilizando o comando `time` do Linux, e portanto são referentes a execução completa do programa, e não só apenas da execução do algoritmo de processamento em si.

5.2.1 Resultados obtidos com o algoritmo Filtro de Lanczos

Nos testes do algoritmo Filtro de Lanczos foi usado como dado de entrada uma matriz de 144 por 73 que contém as informações da temperatura diária de todo o globo entre 1950 e 2011, totalizando aproximadamente 22000 valores. Desse total, foram usados entre 1000 e 21000 valores nos testes realizados.

Nas duas implementações houve uma grande redução do tempo de execução em relação ao tempo da execução sequencial do mesmo, como demonstrado na figura 5.1. O melhor resultado foi obtido na execução da implementação em CUDA na GPU GTX480, a qual obteve um speedup de 20 vezes em relação a execução sequencial numa CPU Intel I7.

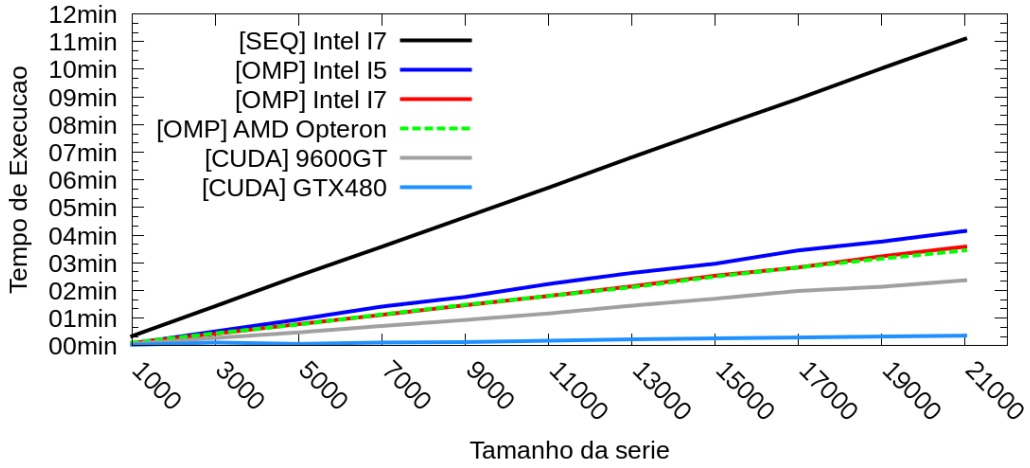

Figura 5.1: Gráfico de tempo do algoritmo Filtro de Lanczos

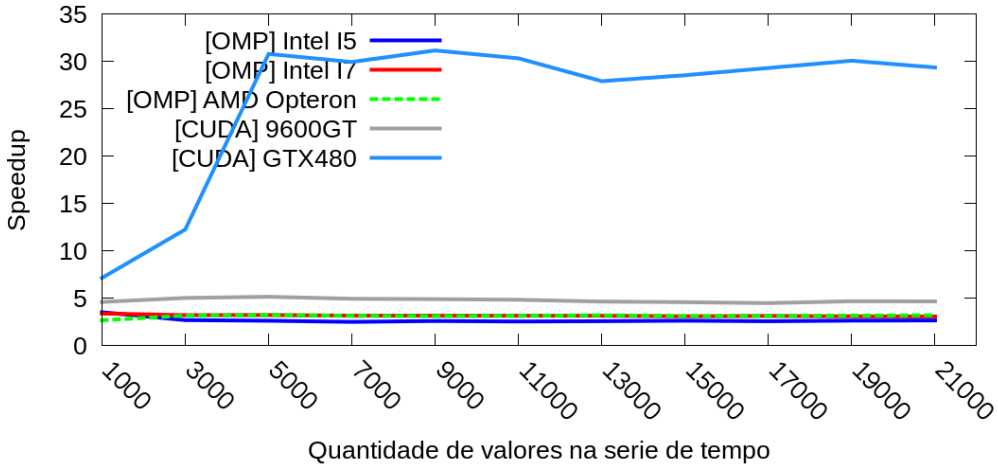

Figura 5.2: Gráfico do speedup do algoritmo Filtro de Lanczos

5.2.2 Resultados obtidos com o algoritmo Teste de Monte Carlo

Nos testes do algoritmo de Monte Carlo foram usados como dados de entrada uma matriz de 48 por 56 que contém as informações da quantidade de chuva diária na América do Sul entre os anos de 1979 e 1999, totalizando 7665 valores por série, e uma série de chuvas de um local na África do mesmo comprimento. Porém, para questão de comparação, o total de valores usados

nos testes foi variado entre 1000 e 5000. Da mesma forma o número de permutações foi variado entre 1000 e 9000.

Nos testes da implementação em OpenMP houve uma grande surpresa, pois o tempo de execução foi incrementado em relação ao tempo de execução sequencial, especialmente ao elevar o número de permutações, como pode ser visto na figura 5.3. No entanto, nos testes da implementação em CUDA houve uma drástica redução desse tempo, inclusive ao incrementar o número de permutações, como mostrado na figura 5.6.

Uma teoria que explique tais resultados é que pelo fato da implementação em OpenMP prover de apenas uma memória que é compartilhada entre todas as threads, ao se fazer as permutações das séries são realizadas inúmeras requisições simultâneas de escrita na memória que por sua vez não é capaz de responder todas elas imediatamente. Esse atraso faz com que as threads fiquem ociosas durante um longo tempo até que todas as requisições sejam executadas. Ao aumentar o número de permutações o número de requisições é proporcionalmente incrementado agravando o problema.

Tal problema não ocorre na implementação em CUDA pois a mesma é executada na GPU, que possui uma arquitetura de memória totalmente diferente a qual provêm a cada thread diversos níveis de memória, incluindo uma própria, como descrito em (NVIDIA..., 2012). Dessa forma as requisições de escrita são distribuídas entre várias memórias independentes reduzindo o congestionamento de requisições.

Para reforçar essa hipótese foi feito alguns testes reduzindo o número de threads usadas na execução da implementação em OpenMP. Esses resultados estão expostos na figura 5.5. É perceptível que ao reduzir o número de threads o tempo de execução reduz consideravelmente, pois o total de requisições simultâneas também é reduzida e assim amenizando o problema.

Esses resultados são importantes para demonstrar que a arquitetura atual dos computadores não é a ideal para se executar algoritmos paralelos.

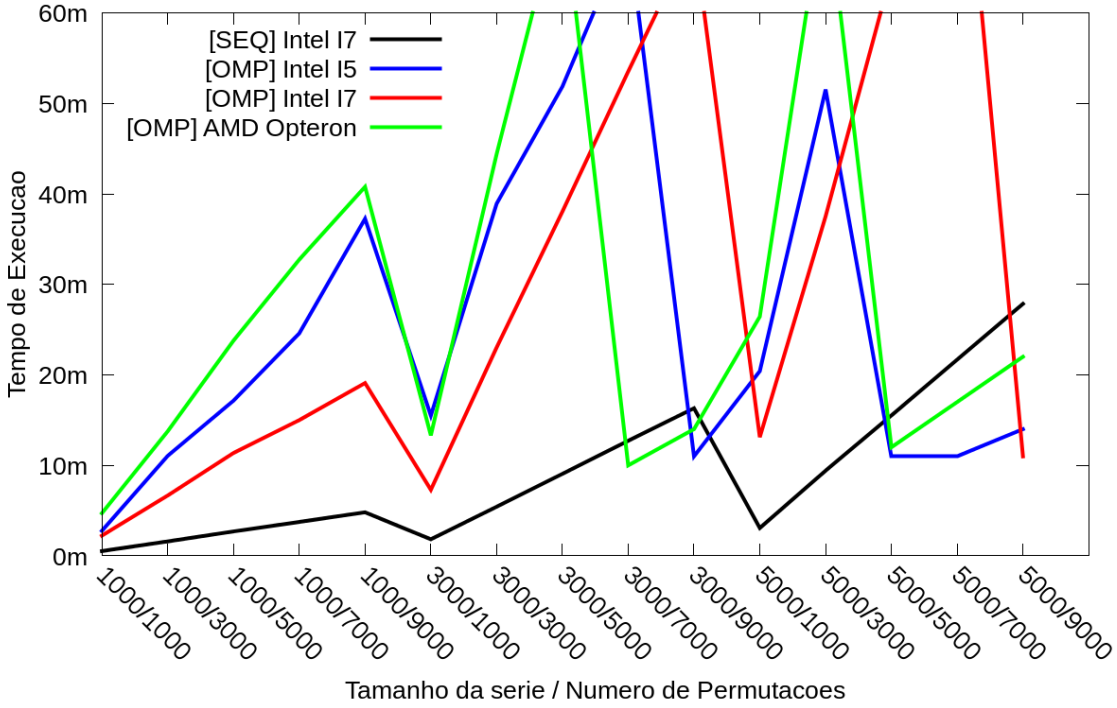

Figura 5.3: Gráfico de tempo do algoritmo Monte Carlo

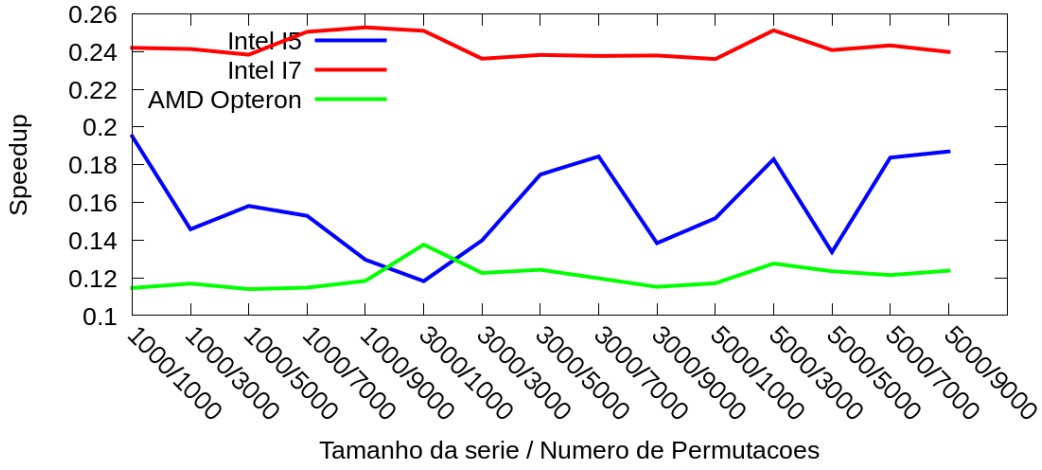

Figura 5.4: Gráfico do speedup do algoritmo Monte Carlo

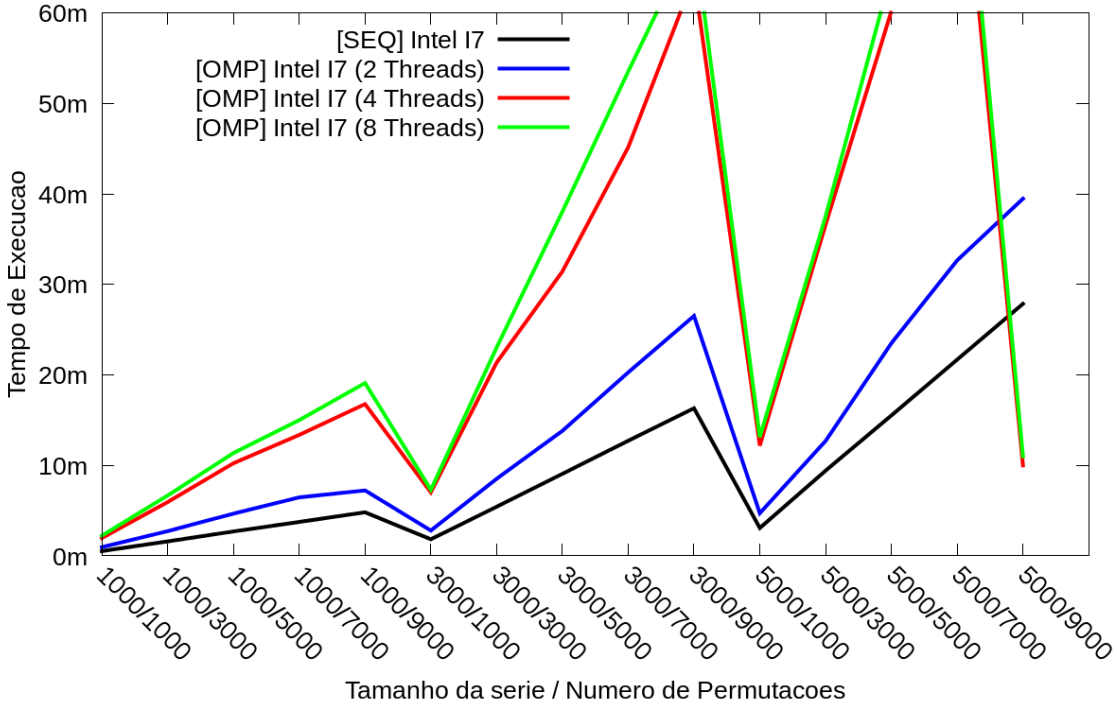

Figura 5.5: Gráfico de tempo do algoritmo Monte Carlo

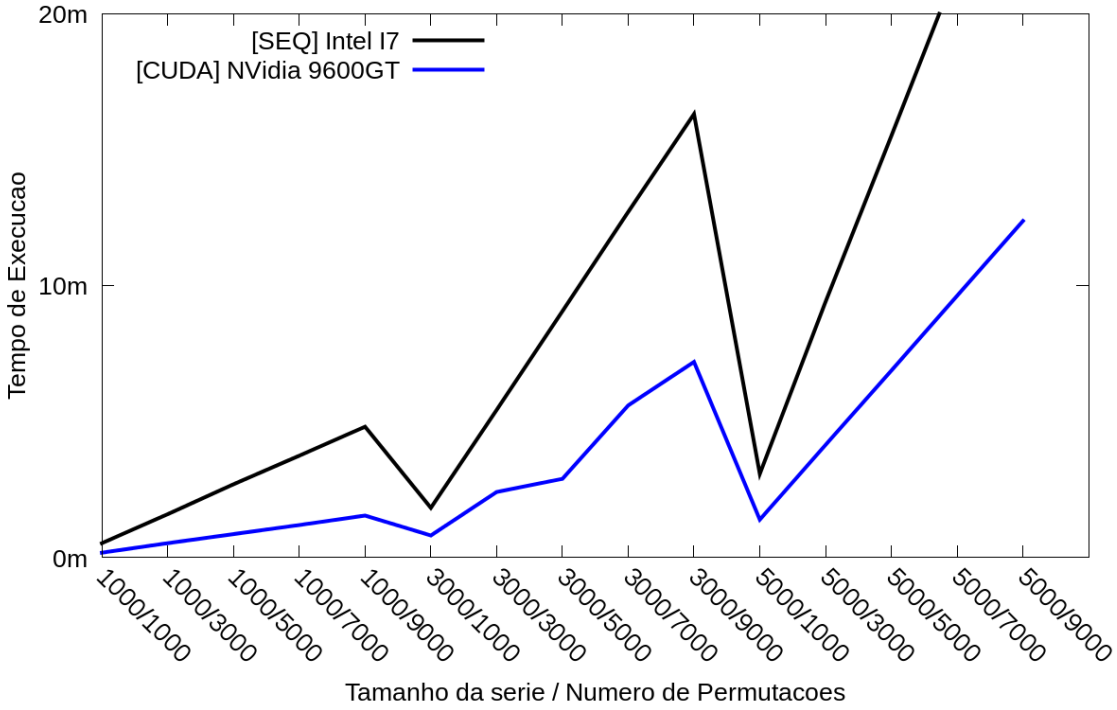

Figura 5.6: Gráfico de tempo do algoritmo Monte Carlo

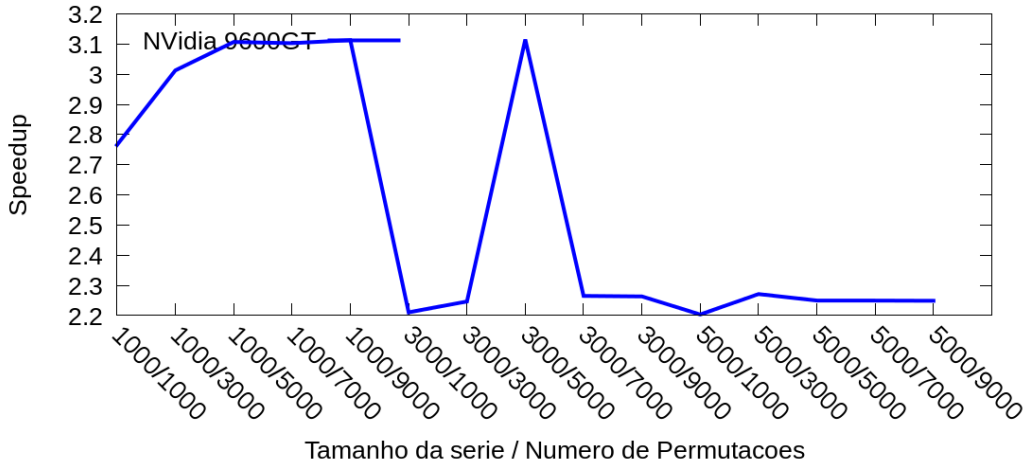

Figura 5.7: Gráfico do speedup do algoritmo Monte Carlo

5.3 Análise dos resultados

Como podemos ver, mesmo sem nenhuma otimização tanto no código quanto nos parâmetros de execução do Kernel, os tempos de execução da versão CUDA são significantemente menores, apresentando tempos 6 vezes menores quando usado a GPU 9600GT, e de até 40 vezes com a GPU GTX480 nos testes feitos. Além disso, é perceptível pelas curvas apresentadas no gráfico que essa redução será ainda maior a medida que o tamanho da entrada aumentar, comprovando o poder de processamento das GPUs Nvidia com a tecnologia CUDA.

6 *Proposta de um esqueleto de código padrão*

Por fim, resumindo-se tudo o que foi já foi dito e usando como base a implementação em CUDA, que se mostrou a mais eficiente, é apresentado um esqueleto padrão de código que soluciona o problema proposto de forma simplificada. Esse esqueleto visa facilitar o trabalho de adaptação de algoritmos sequenciais de análise e processamento de dados atmosféricos em uma solução paralela do mesmo.

A implementação contém duas estruturas de variáveis, definidas no arquivo 6.1, nomeadas `parametros` e `parametros_exec` que armazenam, respectivamente, os parâmetros de entrada e os parâmetros de execução da função em CUDA. Por sua vez, os parâmetros de execução são calculados pela função `calcula_parametros_execucao`, que tem como argumentos o número de pontos e o tamanho das séries do dado de entrada. Os parâmetros de entrada são lidos pela função `le_parametros_entrada`.

Os dados de entrada são lidos e reorganizados, da forma descrita no capitulo 4.2.1, pela função `le_matriz_entrada` e são apontados pela variável `d_entrada`. Após a leitura dos dados, inicia-se o processo de inicialização e configuração do device para que o mesmo seja capaz de executar as tarefas necessárias. Esse processo começa com a alocação de espaço na memória do device para armazenar os dados de entrada e saída, igualmente como é feito em qualquer outro programa, e para isso é utilizado a função `cudaMalloc` da biblioteca CUDA, a qual se assemelha muito com a função `malloc`. A grande diferença desse passo para o que normalmente é feito é que o espaço alocado não é para armazenar todo o dado de entrada mas apenas uma parte desse referente ao que será processado em cada ciclo.

Após esse passo inicia-se o laço de ciclos. Em cada um desses ciclos serão executados os passos descritos na seção 4.2.2. Para copiar os dados entre o host e o device é usado a função `cudaMemcpy`. Para executar o algoritmo em si é necessário passar como parâmetro, além dos parâmetros usuais, a quantidade de blocos por grid e de threads por bloco. Essas informações serão usadas pelo device para organizar as threads a serem executadas dentro hard-

ware disponível no device. O número de threads por bloco é definida no início do arquivo 6.1 como uma constante de nome `THREADS_POR_BLOCO`. A alteração desse valor pode trazer uma melhora de desempenho, no entanto depende diretamente do device em que o código será executado. Para facilitar o uso do código foi escolhido um valor padrão com o qual foi obtido em média os melhores resultados em todos os devices testados. Já o número de threads por bloco é calculada pela função `calcula_parametros_execucao` levando em conta o número total de blocos e a quantidade de pontos a serem processados a cada ciclo. Por fim, a função `cudaDeviceSynchronize` garante que a execução do código principal só continue quando o todo o processamento no device termine para evitar que resultados ainda não calculados sejam copiados do device para o host.

Quando todos os ciclos terminarem, os resultados são escritos em disco pela função `salva_arq_saida` no mesmo formato do dado de entrada original e os espaços em memória são liberados.

Código-Fonte 6.1: esqueleto_codigo.h

```
1  //ESTRUTURA PARA ARMAZENAR OS PARAMETROS
2  //RELATIVOS AOS DADOS
3  typedef struct _parametros{
4    char *arq_entrada_1, *arq_entrada_2, *arq_saida;
5    int NP, NT;
6    float UNDEF;
7  } parametros;
8
9  //ESTRUTURA PARA ARMAZENAR OS PARAMETROS
10 //RELATIVOS A EXECUCAO
11 typedef struct _parametros_exec{
12   int total_npos, npos_por_ciclo, total_ciclos,
13     threads_por_bloco, blocos_por_grid;
14   size_t tam_por_ciclo, mem_total, mem_free;
15 } parametros_exec;
16
17 //DEFINICAO DO NUMERO DE THREADS POR BLOCO
18 #define THREADS_POR_BLOCO 16
```

Código-Fonte 6.2: esqueleto_codigo.c

```
1  #include "esqueleto_codigo.h"
2
3  //VARIAVEIS GLOBAIS:
4  parametros param;
5  parametros_exec param_exec;
```

```
6
7  float *h_entrada = NULL;
8  float *h_saida = NULL;
9
10 float *d_entrada = NULL;
11 float *d_saida = NULL;
12
13 void desaloca_variaveis(){
14   if (h_entrada != NULL) free(h_entrada);
15   if (h_saida != NULL) free(h_saida);
16
17   if (d_entrada != NULL) cudaFree(d_entrada);
18   if (d_saida != NULL) cudaFree(d_saida);
19 }
20
21 int calcula_pos_matriz(int NT, int p, int t){
22   return (p*NT)+t;
23 }
24
25 void le_argumentos(int argc, char **argv, parametros *param) {
26   if (argc-1 < 7){
27     printf("PARAMETROS INVALIDOS!\n");
28     exit(1);
29   }
30   param->arq_entrada_1=argv[1];
31   param->arq_entrada_2=argv[2];
32   param->arq_saida=argv[3];
33   param->NP=atoi(argv[4])*atoi(argv[5]);
34   param->NT=atoi(argv[6]);
35   param->UNDEF=atof(argv[7]);
36 }
37
38 void le_matriz_entrada(char *arq_entrada, parametros param, float **d){
39   int p,t,pos;
40   FILE *arq;
41   float *buffer;
42
43   arq=fopen(arq_entrada,"rb");
44   if (!arq){
45     printf("Erro na abertura do arquivo de entrada!");
46     desaloca_variaveis();
47     exit (1);
48   }
```

```
49
50    buffer=(float*)malloc(param.NP*sizeof(float));
51
52    (*d)=(float*)malloc((param.NP*param.NT)*sizeof(float));
53
54    for (t=0;t<param.NT;t++){
55      fread(buffer,sizeof(float),param.NP,arq);
56
57      for(p=0;p<param.NP;p++){
58        pos=calcula_pos_matriz(param.NT,p,t);
59        (*d)[pos]=buffer[p];
60      }
61    }
62
63    free(buffer);
64
65    fclose(arq);
66 }
67
68 void salva_dados_saida(parametros param, float *s){
69   FILE *arq;
70   int p,t,pos;
71
72   arq=fopen(param.arq_saida,"wb");
73   if (!arq){
74     printf("Erro na abertura do arquivo de saida!");
75     desaloca_variaveis();
76     exit (1);
77   }
78
79   for (t=0;t<param.NT;t++){
80     for (p=0;p<param.NP;p++){
81       pos=calcula_pos_matriz(param.NT,p,t);
82       fwrite((s+pos),sizeof(float),1,arq);
83     }
84   }
85
86   fclose(arq);
87 }
88
89 void calcula_parametros_execucao(parametros_exec *param_exec, int NP, int
     NT){
90
```

```
91    int npos_por_ciclo;
92    int total_ciclos;
93    size_t tam_por_ciclo;
94
95    total_ciclos=1;
96    npos_por_ciclo=NP;
97
98    cudaMemGetInfo(&param_exec->mem_free, &param_exec->mem_total);
99      verifica_erro_cuda("cudaMemGetInfo",-1);
100
101   tam_por_ciclo=(npos_por_ciclo*NT)*sizeof(float);
102   while ( tam_por_ciclo > (param_exec->mem_free-50) ){
103     total_ciclos=total_ciclos+1;
104     npos_por_ciclo=ceil(NP/total_ciclos);
105     tam_por_ciclo=(npos_por_ciclo*NT)*sizeof(float);
106   }
107
108   param_exec->npos_por_ciclo=npos_por_ciclo;
109   param_exec->total_ciclos=total_ciclos;
110   param_exec->tam_por_ciclo=tam_por_ciclo;
111   param_exec->threads_por_bloco=THREADS_POR_BLOCO;
112   param_exec->blocos_por_grid=(npos_por_ciclo+THREADS_POR_BLOCO-1)/
        THREADS_POR_BLOCO;
113
114   //CALCULA O NUMERO TOTAL DE POSICOES QUE SERAO CALCULADAS
115   // ESSE NUMERO PROVAVELMENTE SERA MAIOR QUE O NP,
116   // POR ISSO NAO SE PODE ALOCAR EXATAMENTE O TAMANHO DA SAIDA ESPERADO.
117   // TEM QUE ALOCAR ESSA MEMORIA A MAIS PARA GARANTIR
118   // QUE NO ULTIMO CICLO NAO SEJA COPIADO DADOS ALEM DO QUE FOI ALOCADO
119   param_exec->total_npos=param_exec->npos_por_ciclo*param_exec->
        total_ciclos;
120
121   return;
122 }
123
124 __global__ void nome_funcao( <ARGUMENTOS> ){
125
126 int p = (blockDim.x * blockIdx.x) + threadIdx.x;
127 if (p >= total_pos) return;
128
129 int ini_seq=(p*NT);
130
131 <ALGORITMO>
```

```
132
133 }
134
135 int main(int argc, char **argv){
136
137 //VARIAVEIS AUXILIARES:
138 int ciclo;
139 int pos_inicio_copia_entrada, pos_inicio_copia_saida;
140
141 //LE OS ARGUMENTOS E OS DADOS DE ENTRADA
142 le_argumentos(argc,argv,&param);
143 le_dados_entrada(param,&h_entrada);
144
145 //*****************************************************//
146 //EXECUCAO CUDA
147
148 //RESETA O DISPOSITIVO CUDA
149 cudaDeviceReset();
150
151 //CALCULA OS PARAMETROS DE EXECUCAO
152 calcula_parametros_execucao(param, &param_exec);
153
154 //ALOCA E COPIA O VETOR "WT"
155 cudaMalloc((void**)&d_wt, param_exec.mem_aux_por_ciclo);
156 cudaMemcpy(d_wt, h_wt, param_exec.mem_aux_por_ciclo,
157             cudaMemcpyHostToDevice);
158
159 //ALOCA O ESPACO PARA OS DADOS DE ENTRADA E SAIDA NO DEVICE
160 cudaMalloc((void**)&d_entrada,param_exec.mem_entrada_por_ciclo);
161 cudaMalloc((void**)&d_saida, param_exec.mem_saida_por_ciclo);
162
163 //ALOCA NO HOST O ESPACO PARA A SAIDA (TOTAL)
164 h_saida=(float*)malloc(param_exec.mem_saida_total);
165
166 for (ciclo=0;ciclo<param_exec.total_ciclos;ciclo++){
167
168   //CALCULA A POSICAO DE INICIO DA COPIA DOS DADOS DE ENTRADA
169   pos_inicio_copia_entrada=(ciclo*param_exec.npos_por_ciclo)*param.NT;
170
171   //COPIA OS DADOS DE ENTRADA
172   cudaMemcpy(d_entrada,(h_entrada+pos_inicio_copia_entrada),
173       param_exec.mem_entrada_por_ciclo, cudaMemcpyHostToDevice);
174
```

```
175    //EXECUTA O FILTRO LANCZOS EM CUDA
176    nome_funcao<<<param_exec.blocos_por_grid,
177                  param_exec.threads_por_bloco>>>
178                  (<ARGUMENTOS>);
179
180    //ESPERA O TERMINIO DA EXECUCAO DO DEVICE
181    cudaDeviceSynchronize();
182
183    //CALCULA A POSICAO DE INICIO DA COPIA DOS DADOS DE SAIDA
184    pos_inicio_copia_saida=(ciclo*param_exec.npos_por_ciclo)*param.NT;
185
186    //COPIA A SAIDA DO DEVICE PARA O HOST
187    cudaMemcpy((h_saida+pos_inicio_copia_saida),d_saida,
188         param_exec.mem_saida_por_ciclo, cudaMemcpyDeviceToHost);
189 }
190 //*****************************************************//
191
192 salva_arq_saida(param, h_saida);
193
194 desaloca_variaveis();
195
196 return 0;
197 }
```

Conclusão

Podemos concluir com esse trabalho que com apenas algumas modificações, necessárias para incluir as premissas básicas para que o programa execute em GPUs Nvidia capacitadas com a tecnologia CUDA, podemos reduzir drasticamente o tempo de execução de algoritmos de processamento e análise de dados atmosféricos sem abrir mão da confiabilidade dos resultados. Além disso, o trabalho demonstra que tal método de conversão possui um provável potencial de se adaptar a outros algoritmos que possuem características parecidas com o demonstrado, o que facilita ainda mais a conversão destes.

Referências Bibliográficas

DUCHON, C. E. Lanczos filtering in one and two dimensions. *Journal of Applied Meteorology*, 1979.

MET Office Hadley Centre. `http://www.metoffice.gov.uk/hadobs/hadisst/index.html`. Acessado: 27/03/2013.

NASA. `http://precip.gsfc.nasa.gov/`. Acessado: 27/03/2013.

NOAA. `http://www.esrl.noaa.gov/psd/data/`. Acessado: 27/03/2013.

NVIDIA CUDA C Programming Guide, Version 4.2. 2012.

SABOIA, J. P. J. *Variabilidade interdecadal de precipitação na América do Sul: Característica, impactos e mecanismos*. Dissertação (Mestrado) — Universidade Federal do Paraná, 2010.