Bruno Nocera Zanette

Conversão de algoritmos de processamento de dados meteorológicos de C para CUDA

Curitiba, Março de 2013

Bruno Nocera Zanette

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, Março de 2013

Sumário

Lista de Figuras

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 tal tarefa traz um grande ganho de desempenho sem exigir um conhecimento prévio em programação paralela por parte do programador. Tal objetivo foi pensado especialmente para que os resultados do trabalho possam servir como uma introdução a programação paralela baseada na arquitetura CUDA para qualquer pessoa envolvida em estudos na área atmosférica, desde um pesquisador mais experiente até o mais novo aluno de iniciação científica. O trabalho tem como conclusão um modelo de código genérico que pode ser usado como base para a conversão de novos algoritmos, e demonstra durante o desenvolvimento os problemas e soluções encontrados durante o processo de escrita desse código, além de detalhar as principais características dos algoritmos e dados de entrada usados como base para estudos atmosféricos.

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[1] 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 eficiente, fazendo com que algoritmos simples tenham um tempo elevado de execução, e assim desperdiçando o hardware disponível.

Nesse trabalho serão estudados 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 capítulos 2 e 3 serão descritos, respectivamente, os algoritmos utilizados como base para o estudo e a organização dos dados usados como entrada para esses algoritmos. No capítulo 4 serão detalhadas as implementações paralelas em OpenMP e em CUDA desses algoritmos. Os resultados dos testes dessas implementações serão demonstrados no capítulo 5. Por fim, no capítulo 6, será apresentado um modelo de código genérico que resolve o problema proposto de

[1] Centro de Previsão de Tempo e Estudo Climáticos / Instituto Nacional de Pesquisas Espaciais

maneira simples e que pode ser usado para a conversão de outros algoritmos semelhantes.

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 relacionados à 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 (1), 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:

$$Para\ (t = 1\ ,\ NT):\ Res[t + \lfloor \frac{K}{2} \rfloor] = \sum_{i=0}^{K-1} (WT[i] \times Dados[t+i])$$

Para cada resultado produzido por esse algoritmo são necessárias K somas e multiplicações, onde K é o número de pesos usados, definidos por (1). 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 uma função sequencial 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.

O algoritmo a ser estudado nesse trabalho foi retirado de (2) e é dado 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;[1]
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;

O tempo de 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 de cálculos de novos coeficientes de correlação. Sabendo-se que essas séries podem conter centenas de valores, a quantidade de acessos a memória necessários para escrever e ler todos esses valores é muito grande, elevando o tempo de execução consideravelmente.

[1] O cálculo da correlação é linear, e exige aproximadamente $NT * 10$ operações, onde NT é a quantidade de valores em cada série

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 medidas uma ou mais vezes por dia. A partir da obtenção dessas séries de dados em diversos locais espalhados por uma determinada área, ou até mesmo no globo terrestre inteiro, são feitas verificações para a validação desses valores e executados 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 criada 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.

Em geral 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 pode 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[1] (3) e a NASA[2] (5) dos Estados Unidos da América, e o Met Office Hadley Centre (4) do Reino Unido. São dados como esses que servem como base para estudos sobre mudanças climáticas.

[1] National Oceanic & Atmospheric Administration
[2] National Aeronautics and Space Administration

3.1 Estrutura de dados

A estrutura de dados usada para armazenar essas informações é uma matriz tridimensional (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 fixa 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. A figura abaixo exemplifica essa organização.

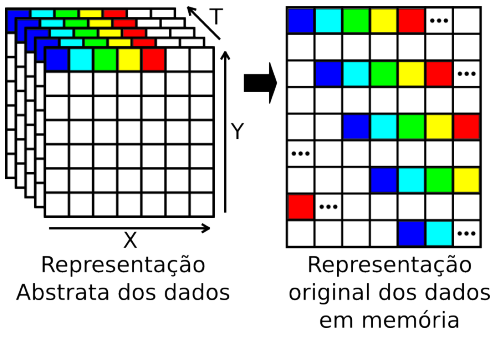

Figura 3.1: Gráfico do tamanho dos arquivos de dados

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 (NP = NX * NY) pelo tamanho de cada uma das séries de dados (NT). Por exemplo, para armazenar as informações referentes 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[3]), ou aproximadamente 90MB. Pode não parecer muito mas, sabendo-se que cada uma dessas quadrículas abrange uma área de aproximadamente 110Km^2 e são as informações de apenas 1 ano, percebe-se que para um período de dados um pouco maior ou para quadrículas com uma área um pouco menor esse valor pode chegar facilmente na casa dos GigaBytes. Prova disso é que, para esse mesmo exemplo citado, caso fosse um período de 50 anos seriam necessários aproximadamente 4.4GB, como demonstrado no gráfico abaixo.

[3]Quantidade de bytes necessária para armazenar um número real

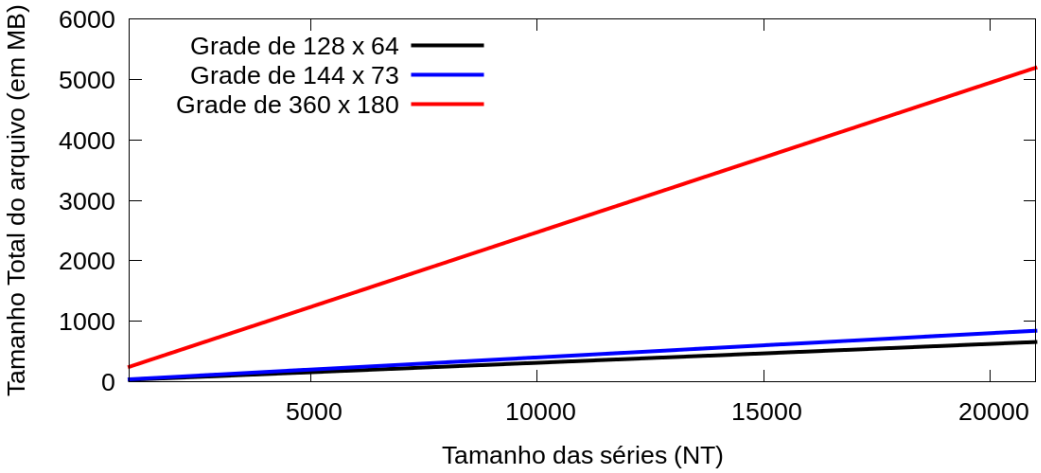

Figura 3.2: Gráfico do tamanho dos arquivos de dados

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 serão analisadas as principais etapas e dificuldades das implementações paralela em OpenMP (6) e CUDA (7) dos algoritmos Filtro de Lanczos e Teste de Monte Carlo, apresentados no capítulo 2.

4.1 Implementação em OpenMP

A maior vantagem em se usar as bibliotecas OpenMP (6) 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 traz 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. Por estas razões essa implementação será usada apenas como referência.

4.2 Implementação em CUDA

Diferente da solução com base em OpenMP, a implementação em CUDA (7) 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.

CUDA é uma arquitetura criada pela empresa NVidia que utiliza placas-de-vídeo para executar algoritmos em paralelo, utilizando as centenas de núcleos de processamento presentes nessas placas, também chamadas de GPUs. Dentro do contexto CUDA essa placa é chamada de *device* e a máquina em que esse device está acoplado é chamada de *host*.

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, 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, requerem 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 pois impede que os mesmos sejam repartidos, já que os algoritmos exigem as séries completas de cada quadrícula e os valores de cada uma dessas séries estão separados na memória.

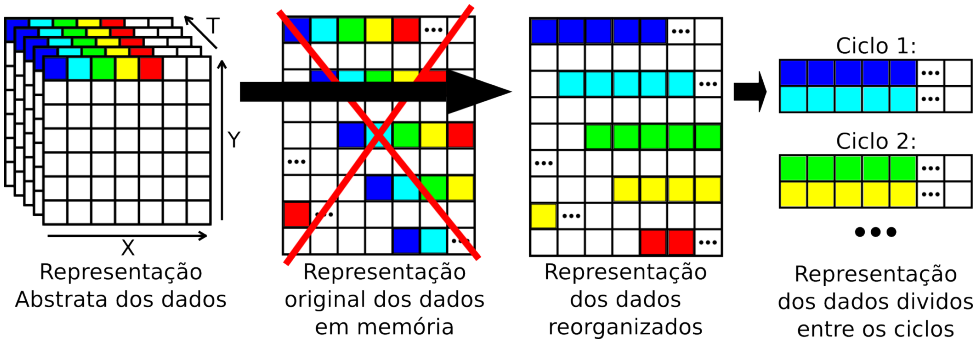

Figura 4.1: Organização dos dados

Para contornar 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, como demonstrado na figura 4.1, assim sendo possível copiar pedaços separados desse dado para o device. Além disso, ao invés da estrutura de matriz, foi usada 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 a que menos precisou de ajustes. Na função que implementa o 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 define qual quadrícula cada um dos núcleos de processamento do device será responsável por processar. Essas alterações estão detalhadas na figura 4.2, juntamente com as necessárias para implementar a solução em OpenMP.

```
__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 4.2: Comparação da função que implementa o algoritmo

Foi a função `Main` do código em C que sofreu as maiores mudanças, como pode ser visto na figura 4.3. 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 no device. 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;[1]
5. Cópia dos dados de saída do Device para o Host;

[1]Esse cálculo depende do tipo de resultado dado pelo algoritmo em questão.

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 feita nenhuma modificação ou otimização no código que implementa os algoritmos em si.

```
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 4.3: Comparação da função `Main`

5 *Resultados*

Para analisar o tempo de execução real das duas implementações é preciso primeiramente verificar se os resultados obtidos são corretos, e posteriormente verificar o quão grande foi a redução do tempo de execução comparado a versão sequencial dos algoritmos. Nesse capítulo serão estudados 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 os mesmos estão certos ou errados. 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 das implementações em OpenMP e 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. 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 núcleos	8	4	8

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

Tabela 5.2: Modelos GPU

	9600GT	**GTX480**	**Tesla C1060**	**Tesla C2050**
Clock	1.96 GHz	1.4 GHz	1.3 GHz	1.15 GHz
Total Memória	512 MB	1536 MB	4096 MB	2687 MB
Clock da Memória	900 Mhz	1848 Mhz	800 Mhz	1500 Mhz
Núcleos CUDA	64	480	240	448
Threads por bloco	512	1024	512	1024

Além disso, na versão CUDA foi utilizado como parâmetro de execução 16 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 para cada quadrícula. Desse total, foram usados entre 5000 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 nas figuras 5.1 e 5.2. O melhor resultado foi obtido na execução da implementação em CUDA na GPU GTX480, a qual obteve uma melhora de trinta vezes em relação a execução sequencial numa CPU Intel I7. Com a implementação em OpenMP se obteve uma redução de apenas três vezes.

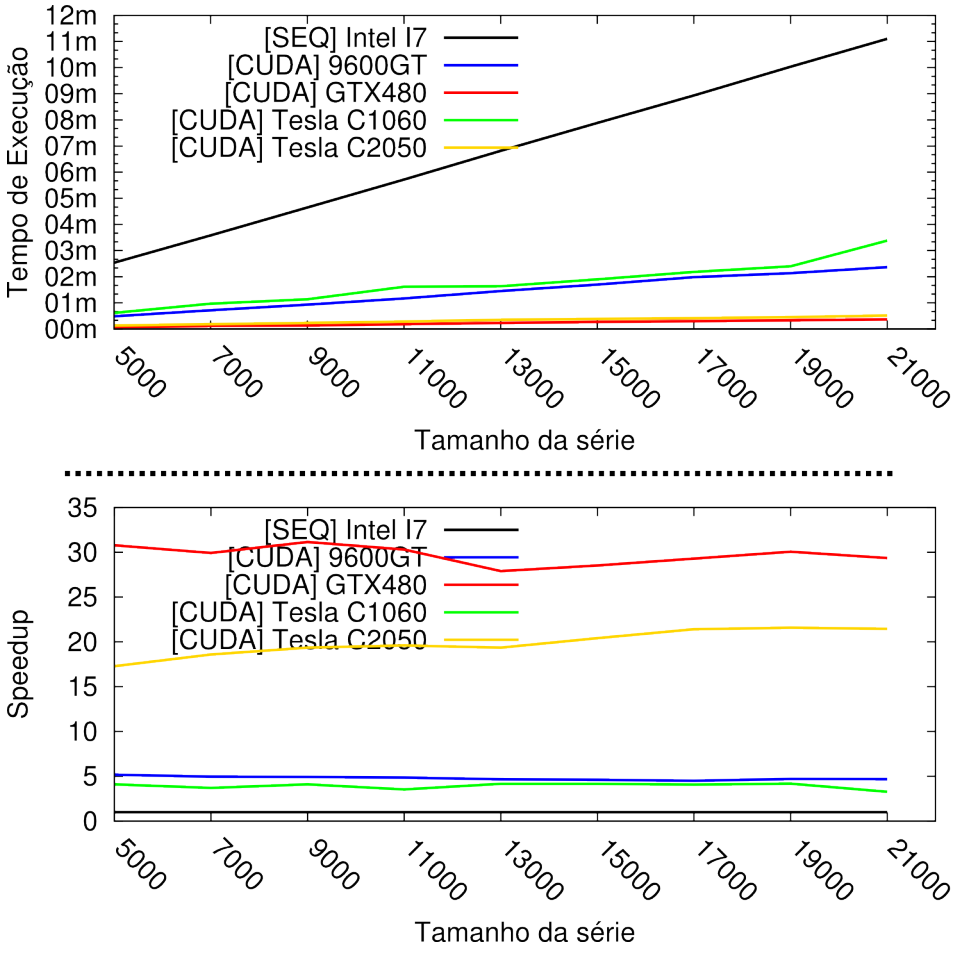

Figura 5.1: Gráficos do tempo de execução e speedup da implementação em CUDA do algoritmo Filtro de Lanczos

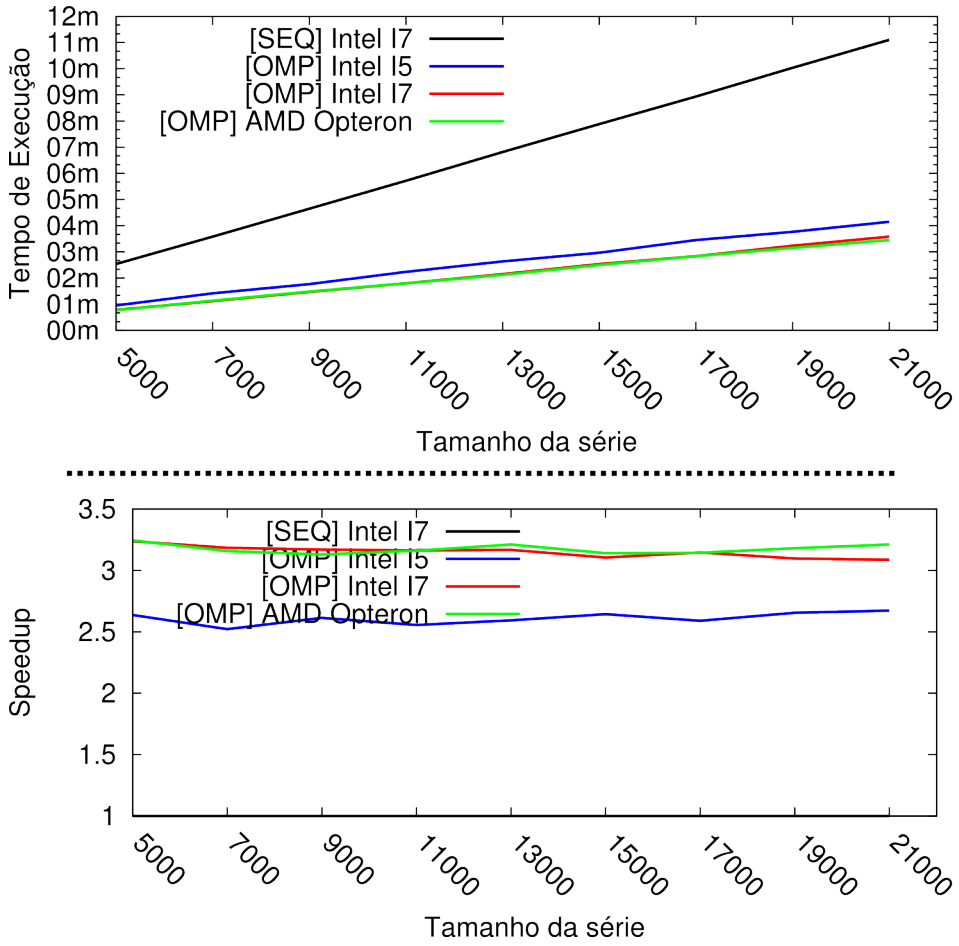

Figura 5.2: Gráficos do tempo de execução e speedup da implementação em OpenMP do algoritmo Filtro de Lanczos

5.2.2 Resultados obtidos com o algoritmo Teste de Monte Carlo

Nos testes do algoritmo de Monte Carlo foram usados como dado 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 chuva 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.

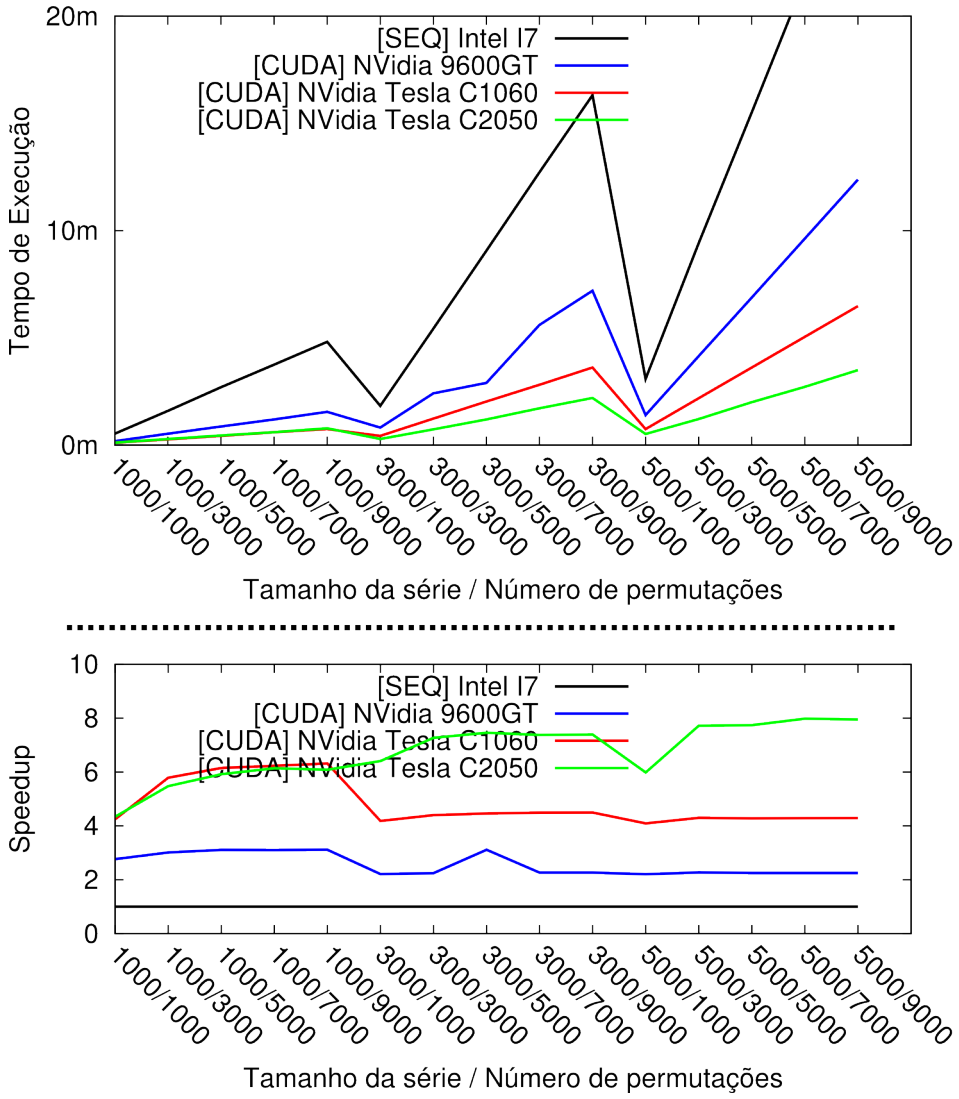

Figura 5.3: Gráficos do tempo de execução e speedup da implementação em CUDA do algoritmo de Monte Carlo

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 5.4. 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 5.3.

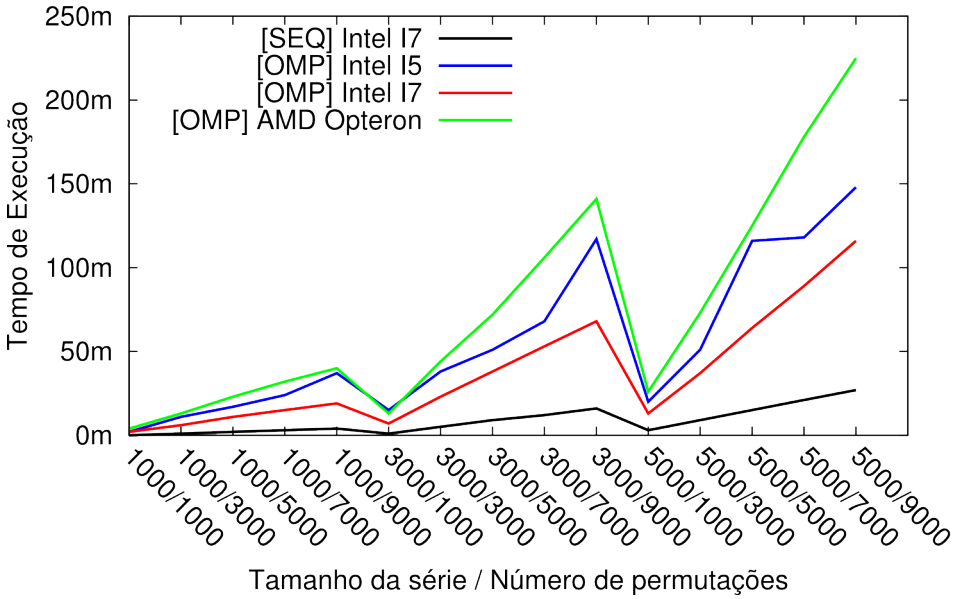

Figura 5.4: Gráfico do tempo de execução da implementação em OpenMP do algoritmo Monte Carlo

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 provê à cada thread diversos níveis de memória, incluindo uma própria, como descrito na seção 2.3 de (7). Dessa forma as requisições de escrita são distribuídas entre várias memórias independentes reduzindo o congestionamento de requisições.

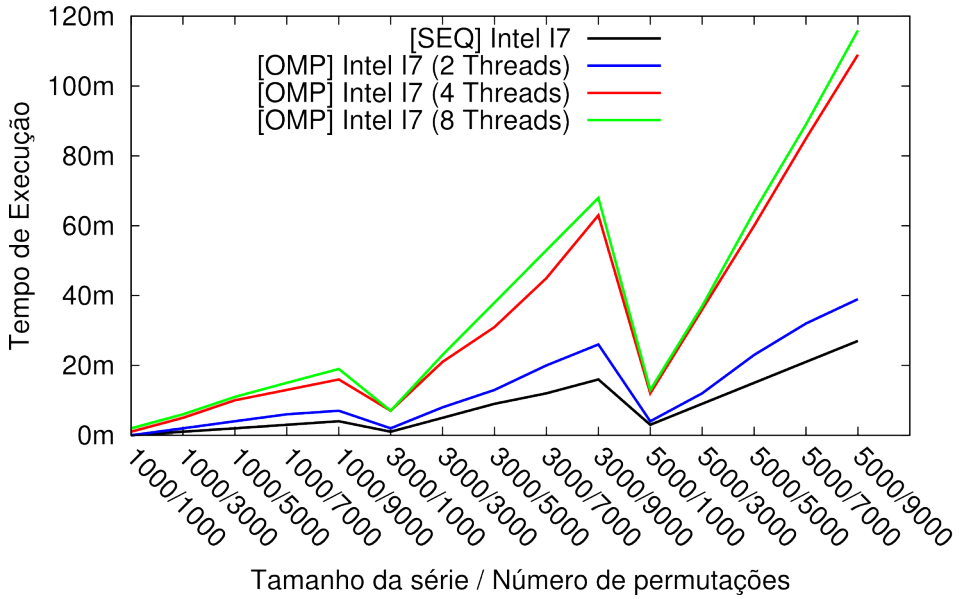

Figura 5.5: Gráfico do tempo de execução da implementação em OpenMP do algoritmo Monte Carlo com a variação do número de threads

Para reforçar essa hipótese foram feitos 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. Ainda assim os tempos de execução continuaram maiores do que a implementação sequencial.

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

5.3 Análise dos resultados

Como podemos ver, mesmo sem nenhuma otimização tanto no código quanto nos parâmetros de execução do código em CUDA, os tempos de execução da versão CUDA são significantemente menores, apresentando tempos seis vezes menores quando usado uma placa-de-vídeo mais antiga como a GPU 9600GT, e de até 30 vezes com a GPU GTX480 nos testes feitos, comparando-as com a CPU Intel I7, considerada por muitos a mais potente da atualidade.

Muitos dos computadores atuais possuem uma placa-de-vídeo capacitada com a tecnologia CUDA. Porém, na grande maioria das vezes, esse recurso é realmente requisitado apenas para visualização de vídeos e jogos. Sabendo disso os resultados desse trabalho demonstram que esse recurso poderia ser melhor aproveitado para fins acadêmicos.

6 *Proposta de um esqueleto de código padrão*

Por fim, resumindo-se tudo o que foi já foi exposto 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 processamento de dados atmosféricos em uma solução paralela do mesmo.

A implementação contém duas estruturas de variáveis, definidas no código-fonte 6.1, nomeadas `parametros` e `parametros_exec` que armazenam, respectivamente, os parâmetros de entrada e os parâmetros de execução da função em CUDA.

Código-Fonte 6.1: Estrutura de Dados

```
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;
```

A função que será executada no device não difere muito da implementação sequencial, como já foi demonstrado na figura 4.2. Primeiramente é necessário adicionar a declaração `__global__` antes do nome da função, para que o compilador saiba que essa será a função a ser

executada pelo device. Depois é preciso definir qual quadrícula cada thread processará, verificar se essa quadrícula está dentro do escopo do ciclo ou não e por fim calcular a posição do vetor que contém o início dos dados da quadrícula a ser processada. Essas três etapas estão definidas nas linhas 3, 4 e 6 do código-fonte 6.2. Por fim há a implementação do algoritmo em si. Durante essa etapa só é necessário alguma alteração se o algoritmo possuir alguma função que não seja compatível com a arquitetura CUDA. Nesse caso é preciso encontrar uma função equivalente nas bibliotecas CUDA.

Código-Fonte 6.2: Função que implementa o algoritmo

```
1  __global__ void nome_algoritmo( <ARGUMENTOS> ){
2
3  int p = (blockDim.x * blockIdx.x) + threadIdx.x;
4  if (p >= total_pos) return;
5
6  int ini_seq=(p*NT);
7
8  <ALGORITMO>
9
10 }
```

Na função `main`, definida no código-fonte 6.3, a primeira coisa a se fazer é ler os argumentos e os dados de entrada e calcular parâmetros de execução.

Os parâmetros de entrada são lidos pela função `le_parametros_entrada`. Os dados são lidos e reorganizados, da forma descrita no capitulo 4.2.1, pela função `le_matriz_entrada` e são apontados pela variável `d_entrada`. Já 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. Essas funções estão definidas respectivamente nas linhas 14, 23 e 46 do código-fonte 6.4.

Após essas etapas é preciso inicializar e configurar o 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.

Logo após esse processo inicia-se o laço de ciclos. Em cada um desses ciclos serão executados os passos descritos na seção 4.2.2. Os passos estão definidos em ordem nas linhas 20, 22,

25, 30 e 32 do código-fonte 6.3.

Código-Fonte 6.3: Função `main`

```
1  int main(int argc, char **argv){
2
3  //VARIAVEIS AUXILIARES:
4  int ciclo, pos_inicio_copia_entrada, pos_inicio_copia_saida;
5
6  le_argumentos(argc,argv,&param);
7  le_dados_entrada(param,&h_entrada);
8
9  calcula_parametros_execucao(param, &param_exec);
10
11 cudaDeviceReset();
12
13 cudaMalloc((void**)&d_entrada,param_exec.mem_entrada_por_ciclo);
14 cudaMalloc((void**)&d_saida, param_exec.mem_saida_por_ciclo);
15
16 h_saida=(float*)malloc(param_exec.mem_saida_total);
17
18 for (ciclo=0;ciclo<param_exec.total_ciclos;ciclo++){
19
20   pos_inicio_copia_entrada=(ciclo*param_exec.npos_por_ciclo)*param.NT;
21
22   cudaMemcpy(d_entrada,(h_entrada+pos_inicio_copia_entrada),
23        param_exec.mem_entrada_por_ciclo, cudaMemcpyHostToDevice);
24
25   nome_algoritmo <<<param_exec.blocos_por_grid,
26            param_exec.threads_por_bloco>>>
27            ( <ARGUMENTOS> );
28   cudaDeviceSynchronize();
29
30   pos_inicio_copia_saida=(ciclo*param_exec.npos_por_ciclo)*param.NT;
31
32   cudaMemcpy((h_saida+pos_inicio_copia_saida),d_saida,
33        param_exec.mem_saida_por_ciclo, cudaMemcpyDeviceToHost);
34 }
35
36 salva_arq_saida(param, h_saida);
37 return 0;
38 }
```

O cálculo da posição de memória, tanto para os dados de entrada quanto os saída, é feito

multiplicando-se o ciclo atual, a quantidade de quadrículas a serem processadas em cada ciclo e o tamanho de cada série. A cópia dos dados entre o host e o device é feita com 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 do hardware disponível no device. O número de threads por bloco é definido no início do arquivo 6.4 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. O número de threads por bloco é calculado pela função `calcula_parametros_execucao`, como já descrito anteriormente. 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 6.4: Funções Complementares

```
1  #include "esqueleto_codigo.h"
2
3  //DEFINICAO DO NUMERO DE THREADS POR BLOCO
4  #define THREADS_POR_BLOCO 16
5
6  //VARIAVEIS GLOBAIS:
7  parametros param;
8  parametros_exec param_exec;
9  float *h_entrada = NULL;
10 float *h_saida = NULL;
11 float *d_entrada = NULL;
12 float *d_saida = NULL;
13
14 void le_argumentos(int argc, char **argv, parametros *param) {
15   param->arq_entrada_1=argv[1];
16   param->arq_entrada_2=argv[2];
17   param->arq_saida=argv[3];
18   param->NP=atoi(argv[4])*atoi(argv[5]);
19   param->NT=atoi(argv[6]);
```

```
20    param->UNDEF=atof(argv[7]);
21  }
22
23  void le_matriz_entrada(char *arq_entrada, parametros param, float **d){
24    int p,t,pos;
25    FILE *arq;
26    float *buffer;
27
28    arq=fopen(arq_entrada,"rb");
29
30    buffer=(float*)malloc(param.NP*sizeof(float));
31    (*d)=(float*)malloc((param.NP*param.NT)*sizeof(float));
32
33    for (t=0;t<param.NT;t++){
34      fread(buffer,sizeof(float),param.NP,arq);
35
36      for(p=0;p<param.NP;p++){
37        pos=calcula_pos_matriz(param.NT,p,t);
38        (*d)[pos]=buffer[p];
39      }
40    }
41
42    free(buffer);
43    fclose(arq);
44  }
45
46  void calcula_parametros_execucao(parametros_exec *param_exec, int NP, int
      NT){
47
48    int npos_por_ciclo, total_ciclos;
49    size_t tam_por_ciclo;
50
51    total_ciclos=1;
52    npos_por_ciclo=NP;
53
54    cudaMemGetInfo(&param_exec->mem_free, &param_exec->mem_total);
55
56    //FOI DECREMENTADO UMA QUANTIA DO VALOR TOTAL DE MEMORIA LIVRE
57    //PARA QUE NAO SEJA USADO 100% DA MEMORIA DISPONIVEL, POR
58    //UMA QUESTAO DE ESTABILIDADE. ISSO PODE CAUSAR TRAVAMENTOS.
59    //ESSE VALOR PODE SER ALTERADO.
60    tam_por_ciclo=(npos_por_ciclo*NT)*sizeof(float);
61    while ( tam_por_ciclo > (param_exec->mem_free-50) ){
```

```
62      total_ciclos=total_ciclos+1;
63      npos_por_ciclo=ceil(NP/total_ciclos);
64      tam_por_ciclo=(npos_por_ciclo*NT)*sizeof(float);
65    }
66
67    param_exec->npos_por_ciclo=npos_por_ciclo;
68    param_exec->total_ciclos=total_ciclos;
69    param_exec->tam_por_ciclo=tam_por_ciclo;
70    param_exec->threads_por_bloco=THREADS_POR_BLOCO;
71    param_exec->blocos_por_grid=(npos_por_ciclo+THREADS_POR_BLOCO-1)/
          THREADS_POR_BLOCO;
72
73    //CALCULA O NUMERO TOTAL DE POSICOES QUE SERAO CALCULADAS
74    //ESSE NUMERO PROVAVELMENTE SERA MAIOR QUE O NP,
75    //POR ISSO NAO SE PODE ALOCAR EXATAMENTE O TAMANHO DA SAIDA ESPERADO.
76    //TEM QUE ALOCAR ESSA MEMORIA A MAIS PARA GARANTIR
77    //QUE NO ULTIMO CICLO NAO SEJA COPIADO DADOS ALEM DO QUE FOI ALOCADO
78    param_exec->total_npos=param_exec->npos_por_ciclo*param_exec->
          total_ciclos;
79
80    return;
81  }
82
83  int calcula_pos_matriz(int NT, int p, int t){
84    return (p*NT)+t;
85  }
86
87  void salva_dados_saida(parametros param, float *s){
88    FILE *arq;
89    int p,t,pos;
90
91    arq=fopen(param.arq_saida,"wb");
92
93    for (t=0;t<param.NT;t++){
94      for (p=0;p<param.NP;p++){
95        pos=calcula_pos_matriz(param.NT,p,t);
96        fwrite((s+pos),sizeof(float),1,arq);
97      }
98    }
99
100   fclose(arq);
101 }
```

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 grande potencial de se adaptar a outros algoritmos que possuem características parecidas com o demonstrado, o que facilita ainda mais a conversão destes.

Como principal contribuição para a comunidade acadêmica, principalmente aos pesquisadores da área atmosférica, o trabalho traz um modelo de código genérico completo que resolve o problema proposto e que pode ser usado como base para a conversão de outros algoritmos, além de explicar passo-a-passo o código apresentado, fazendo com que seja possível até mesmo pessoas com pouquíssima experiência em programação a se aventurarem no mundo da programação paralela.

Referências Bibliográficas

1 DUCHON, C. E. Lanczos filtering in one and two dimensions. *Journal of Applied Meteorology*, v. 18, n. 8, p. 1016–1022, 1979.

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

3 PSD Climate and Weather Data. NOAA - National Oceanic & Atmospheric Administration. Disponível em: <http://www.esrl.noaa.gov/psd/data/>.

4 HADISST - Hadley Centre Sea Ice and Sea Surface Temperature data set. Met Office Hadley Centre. Disponível em: <http://www.metoffice.gov.uk/hadobs/hadisst/index.html>.

5 GPCP - Global Precipitation Climatology Project. NASA - National Aeronautics and Space Administration. Disponível em: <http://precip.gsfc.nasa.gov/>.

6 CHAPMAN, B.; JOST, G.; PAS, R. van der. *Using OpenMP : portable shared memory parallel programming*. [S.l.]: Massachusetts Institute of Technology, 2008. 378 p.

7 NVIDIA. *NVIDIA CUDA C Programming Guide*. 4.2. ed. NVIDIA Corporation, 2012. 173 p. Disponível em: <www.nvidia.com>.