ACELERAÇÃO DO MÉTODO DE OTIMIZAÇÃO L-BFGS USANDO TECNOLOGIA CUDA

Eduardo Bomfim Sanseverino

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Nelson Maculan Filho
Sergio Barbosa Villas-Boas

Rio de Janeiro
Dezembro de 2014

ACELERAÇÃO DO MÉTODO DE OTIMIZAÇÃO L-BFGS USANDO TECNOLOGIA CUDA

Eduardo Bomfim Sanseverino

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Nelson Maculan Filho, D.Sc.

Prof. Sergio Barbosa Villas-Boas, Ph.D.

Prof. Adilson Elias Xavier, D.Sc

Prof. Carmen Lucia Tancredo Borges, D.Sc.

Prof. Luiz Satoru Ochi, D.Sc.

RIO DE JANEIRO – RJ, BRASIL
DEZEMBRO DE 2014

Sanseverino, Eduardo Bomfim

Aceleração do método de otimização L-BFGS usando tecnologia CUDA/Eduardo Bomfim Sanseverino. – Rio de Janeiro: UFRJ/COPPE, 2014.

XI, 50 p.: il.; 29, 7cm.

Orientadores: Nelson Maculan Filho e Sergio Barbosa Villas-Boas.

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2014.

Referências Bibliográficas: p. 49 – 50.

1. Otimização não-linear 2. L-BFGS 3. CUDA I. Maculan Filho, Nelson. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título

"Combati o bom combate, acabei a carreira, guardei a fé."(Paulo de Tarso)

Agradecimentos

Agradeço a Deus pela oportunidade de poder estudar em uma das melhores instituições do país e ter me concedido chegar até aqui.

Aos meus avós maternos que me criaram como um filho e me fizeram ser o homem que sou hoje.

A minha esposa Aline que muito me ajudou e incentivou até chegar aqui. Tenho certeza que sem a sua ajuda e incentivo jamais conseguiria.

A minha filha Alice que sempre me motivou com seu sorriso e intensa alegria.

Ao meu filho Gabriel que ainda na barriga me incentiva muito a terminar esse curso.

Ao incansável e prestativo professor Maculan que me aceitou como seu orientando e me guiou por todo esse caminho de estudo sempre me ajudando em todas as áreas da minha vida.

Ao Professor Sérgio Barbosa Villas-Boas que me encorajou a fazer o mestrado e me orientou.

Ao Professor Heraldo Almeida que por diversas vezes parou tudo o que estava fazendo para me atender em minhas questões.

Ao meu amigo Renan Vicente que passou horas comigo no laboratório (LABOTIM) tirando dúvidas.

Ao meu primo-irmão Salomão, uma pessoa que amo muito e sempre acreditou que eu conseguiria terminar o curso. Com palavras de ânimo nos momentos mais difíceis embora tenha ficado com medo de eu ficar maluco de tanto estudar para alcançar esse objetivo.

Agradeço ao povo brasileiro por pagar meu curso de pos-graduação.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre (M.Sc.)

ACELERAÇÃO DO MÉTODO DE OTIMIZAÇÃO L-BFGS USANDO TECNOLOGIA CUDA

Eduardo Bomfim Sanseverino

Dezembro/2014

Orientadores: Nelson Maculan Filho
Sergio Barbosa Villas-Boas
Programa: Engenharia de Sistemas e Computação

Dentro da linha de Otimização o algoritmo de Limited Broyden-Fletcher-Goldfarb-Shanno é um método interativo para resolução de problemas irrestritos de otimização não-linear; seu código fonte é aberto.

Este trabalho analisa e em seguida implementa a versão paralelizada desse algoritmo acelerando-o com o uso de GPU e tecnologia CUDA.

Para o trabalho de escrever o código paralelo, optou-se pelo uso da biblioteca cuBLAS. Dessa forma o esforço de desenvolvimento de software é mitigado, e obtêm-se um código final com alto grau de manutenibilidade, pois desenvolve-se software, descartando a alta complexidade de escrita em código nativo CUDA.

O trabalho foi validado com experimentos computacionais. Os resultados experimentais são analisados, com isso determinam-se algumas condições para ocorrência de *speedup*.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

ACCELERATION OF THE OPTIMIZATION METHOD L-BFGS USING CUDA TECHNOLOGY

Eduardo Bomfim Sanseverino

December/2014

Advisors: Nelson Maculan Filho
Sergio Barbosa Villas-Boas
Department: Systems Engineering and Computer Science

In Optimization Science, the algorithm of Limited Broyden-Fletcher-Goldfarb-Shanno is an interactive method for solving unconstrained nonlinear optimization problems; its source code is open.

This paper analyses and then implements the parallel version of this algorithm accelerating it by using GPU with CUDA technology.

For the task of writing parallel code, it was opted to use cuBLAS library. Thus the effort of software development is mitigated, and the final code is obtained with a high degree of maintainability, for the source code is not written using the highly complex native CUDA code.

The work was validated with computational experiments. The experimental results are analyzed, and it is determined some conditions for the occurrence of speedup.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Revisão Bibliográfica

Apresentação da tecnologia CUDA [19] da NVIDIA. Será mostrada a história da biblioteca BLAS [15] e a sua evolução para CBLAS [15] e posteriormente o paralelismo com *GPU* em cuBLAS [17].

1.1 Computação Paralela

A computação paralela tem como objetivo realizar vários cálculos ao mesmo tempo em diferentes processadores, obtendo assim, maior velocidade de processamento gerando resultados em tempo muito menor. Isso é uma forma interessante de diminuir o tempo de processos que possuem um tamanho grande demais, ocupando bastante tempo uma única central de processamento.

Em verdade, esse tipo de abordagem pode ser comparado a forma com que trabalhamos, tendo uma tarefa complexa demais para ser resolvida, o que fazemos é dividi-la em pedaços menores e mais fáceis de serem concluídos. Dessa forma atingimos o nosso objetivo principal que é a conclusão do todo com a soma de pequenas partes processuais. Em computação paralela é exatamente o que acontece.

Tendo-se uma operação complexa, ela pode ser quebrada em pedaços menores e mais simples que serão processadas por outros núcleos de processamento ou até mesmo outras máquinas que fazem parte de um sistema maior como um *cluster*.

Feito isso, o resultado é obtido de forma muito mais rápida podendo ser vital para aquela tarefa ou operação.

Como podemos observar o uso do paralelismo é apenas uma implementação

computacional daquilo que já vivenciamos em nosso cotidiano com algumas tarefas.

A construção de processadores com vários núcleos foi uma iniciativa da indústria com pouco sucesso evolutivo pois não havia um motivo que desse suporte a esse investimento. O uso de vários núcleos acontecia de forma discreta e com a aplicação muito específica em algumas áreas de trabalho.

Embora o investimento em pesquisa para o desenvolvimento de processadores com vários núcleos tenha acontecido, não se pode comparar com o investimento que as placas gráficas obtiveram, tornando-as visíveis no segmento do paralelismo computacional.

Com o aquecimento da indústria de jogos eletrônicos, os fabricantes de placa gráfica, também chamada de *GPU (Graphics Processing Unit)*, tiveram que acompanhar a enorme demanda de entrega de processamento, cada vez mais rápido e eficiente para este público. A solução encontrada foi a de agregar vários núcleos de processamento na mesma *GPU*, de forma que ela conseguisse entregar toda a velocidade de processamento necessária. Essa demanda só poderia ser suprida com paralelismo desenvolvido dentro de vários núcleos existentes na *GPU*. Com o mercado gerando o forte investimento em desenvolver *hardwares* mais velozes, a alta capacidade de processamento deu origem a grandes centros de pesquisas, com profissionais altamente qualificados, que com excelência produziram *GPU's* com poder de processamento elevado que em alguns casos superam muitos processadores como mostrado em 1.1.

Os fabricantes conseguiram um enorme poder de processamento paralelo dentro das suas *GPU's*. Essa tecnologia tornou-se demasiadamente poderosa para ser aplicada somente na indústria de jogos. Foi nesse momento que houve uma mudança de paradigma, em como as *GPU's* deveriam atuar, deixando de ser exclusivamente para a renderização gráfica dos jogos eletrônicos e sim ser expandida como uma nova plataforma para processamento paralelo no âmbito científico e comercial. Faltava apenas a criação de um meio para se ter acesso a todo esse poder de paralelismo existente nas placas gráficas.

Renomados fabricantes habilitaram as suas poderosas *GPU's* a atuar como mais um conjunto de processadores disponíveis aos desenvolvedores de *software*. A *GPU* deixa de ser específica para processar imagens vetoriais e torna-se concorrente do

processador.

Com o aquecimento do mercado e a alta concorrência entre os fabricantes, as *GPU's* ficaram mais acessíveis. Atualmente grande parte das máquinas vendidas possuem *GPU's* capazes de realizar processamento paralelo. Trazendo assim a tecnologia de processamento paralelo ao alcance de vários usuários. Isso com certeza colaborou muito para a divulgação da tecnologia no meio comercial e acadêmico.

O mundo do desenvolvimento de *software* passou a considerar a *GPU* como uma forte aliada na busca de *speedup*. Os fabricantes de *software* passaram a implementar a versão paralela do seu *software*, obtendo assim, uma enorme aceleração no processamento.

1.2 OpenCL

OpenCL é um padrão aberto, mantido pelo *Khronos Group* [7], que permite o uso de *GPU* para desenvolvimento de aplicações paralelas. Ele também permite que os desenvolvedores escrevam códigos de programação heterogêneos, fazendo com que estes programas consigam aproveitar tanto os recursos de processamento das *CPU's* quanto das *GPU's*. Além disso, permite programação paralela usando paralelismo de dados e de tarefas.

O *OpenCL* permite o desenvolvimento de código paralelo independente do fabricante de *GPU*. Isso permitiu maior liberdade no desenvolvimento de aplicações paralelas.

Embora ainda com muito horizonte para ser explorado o *openCL* continua sendo uma opção inferior a tecnologia *CUDA*, diversos trabalhos mostram que a comparação de desempenho entre as duas tecnologias ainda é significante [11].

1.3 Tecnologia CUDA

A NVIDIA, uma das maiores fabricantes de *GPU*, saiu na frente lançando em novembro de 2006 sua plataforma de computação paralela chamada CUDA *(Compute Unified Device Architeture)*, visando disponibilizar os processadores de sua *GPU* para atuar como um novo conjunto de processadores sendo capaz de realizar operações matemáticas em cada um desses núcleos.

Essa tecnologia disponibiliza ao desenvolvedor uma plataforma de computação paralela e um modelo de programação inventado pela NVIDIA. Ela permite aumentos significativos de performance computacional ao aproveitar a potência da *(GPU)*.

O código CUDA é escrito em linguagem *.cu, privada da NVIDIA. Escrever um código nessa linguagem é uma tarefa com alto grau de complexidade, pois é preciso levam em conta conceitos como *cache* do bloco, *cache* do *grid*, *cache* da *thread*, problemas derivados de código divergente (não idêntico dentro do mesmo *warp*, que é um sub-bloco) entre outros. Escrever *software* em linguagem nativa *.cu é portanto complexo e de difícil manutenção.

Em outra visão, a *GPU*, passa a ser um conjunto de processadores disponíveis para executar códigos, aumentando significativamente o desempenho do sistema.

Os jogos de computadores se beneficiam dessa tecnologia para o cálculo de seus polígonos, efeitos em 3D, cálculo de colisões, entre outros processamentos pesados exigidos pela indústria de jogos. Contudo a tecnologia também é apta a trabalhar com aplicações não gráficas.

A *GPU* tornou-se mais eficaz para trabalhar em processamento paralelo, por sua origem ter sido a de atender a demanda da indústria de jogos e atuar massivamente em cálculos em 3D e com o decorrer da história, ela mostrou-se muito eficiente na manipulação de grandes dados.

Existem diferenças de capacidade em cada placa NVIDIA, sendo separadas por versão de capacidade de computação. Placas mais avançadas possuem especificações para se trabalhar com cálculos em pontos flutuantes diferentes das versões iniciais, conforme é mostrado na figura 1.1.

Atualmente a NVIDIA não só comercializa *GPU's* convencionais voltadas para o mercado de jogos eletrônicos mas também *GPU's* de altíssimo desempenho sem saída gráfica. Sendo essas usadas somente como conjunto extra de processadores através da tecnologia CUDA, conforme é mostrado na figura 1.2.

Várias organizações possuem máquinas específicas equipadas com *GPU's* para processamento paralelo. Está cada vez mais comum o uso desse tipo de arquitetura computacional para se obter grande desempenho de paralelismo, conforme é mostrado na figura 1.3.

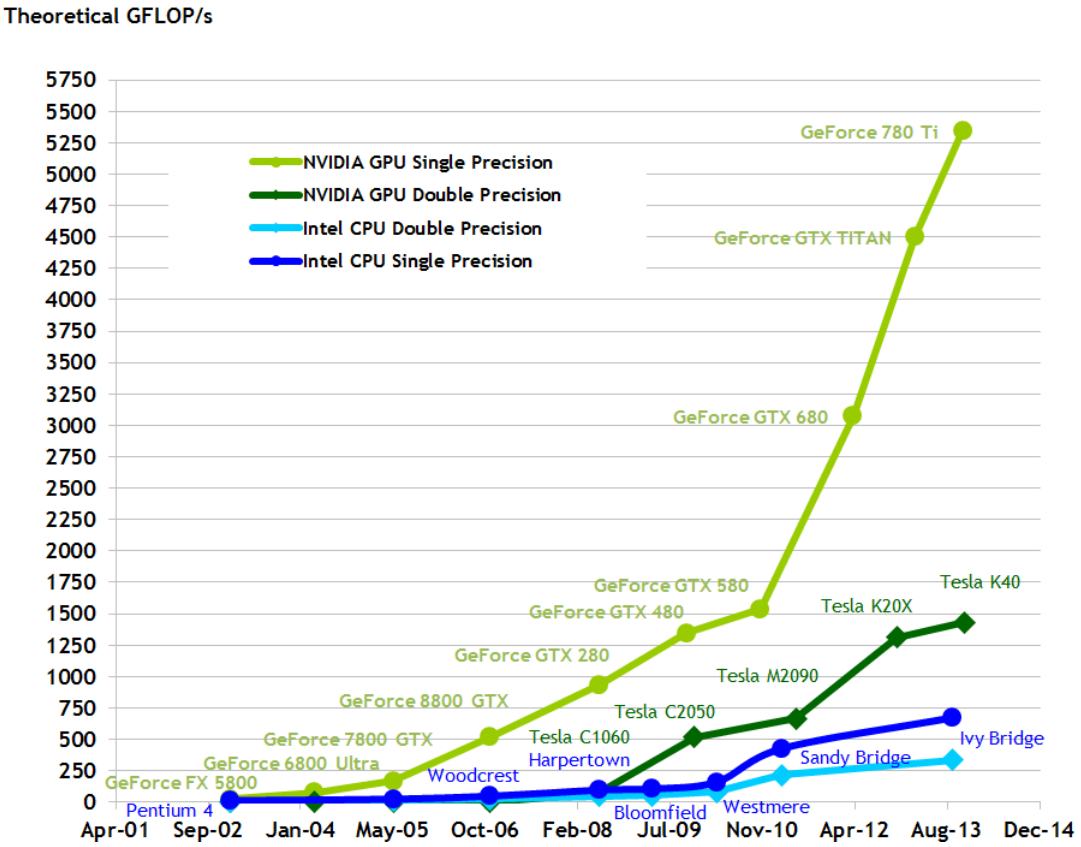

Figura 1.1: *CPU* x *GPU*.

Figura 1.2: NVIDIA *GPU* Tesla.

Figura 1.3: Cuda Tesla Array.

1.4 Arquitetura CUDA.

As *GPU's* com tecnologia CUDA possuem um tipo peculiar de arquitetura para produzir o processamento paralelo. A *GPU* é construída com um conjunto de *Streaming Multiprocessors* (SM), conforme é mostrado na figura 1.4.

Em cada *SM* possui 32 *cuda-cores*. São nos *cuda-cores*, também chamado de *threads* que é realizado o processamento paralelo.

Um programa *MultiThread* é aquele que executa processamento em várias "hardware-threads"simultaneamente.

O dimensionamento de *threads* é feito da seguinte forma, é formado um *grid* de blocos que é subdivido em *threads*, conforme é mostrado na figura 1.5.

Os blocos por sua vez são compostos por várias *threads*, e como existem centenas sendo processadas ao mesmo tempo, temos o modelo paralelo de processamento. Por um cálculo simples cada *thread* tem um único número de identificação, ou seja, cada *thread* será responsável por um único cálculo dentro da natureza do problema que estamos resolvendo, conforme é mostrado na figura 1.6.

Cada bloco tem uma quantidade de memória compartilhada *(shared memory)* que somente é acessível por ele. Cada bloco também tem seu conjunto de registradores que se torna específico para seu conjunto de *threads* locais. Cada bloco compartilha a memória global. Todas as *threads* possuem acesso a memória global,

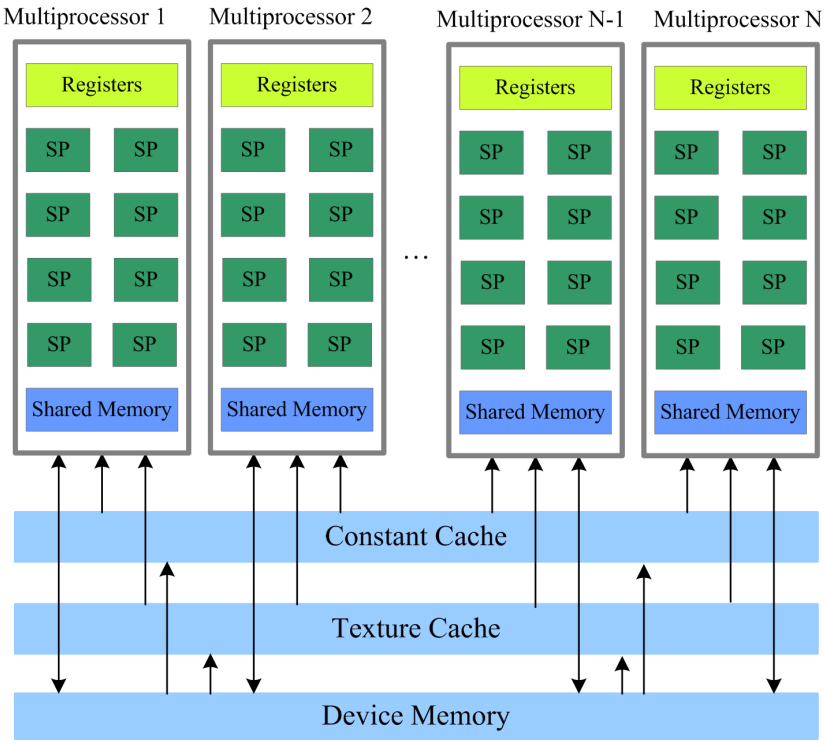

Figura 1.4: Cuda *Streaming Multiprocessors*.

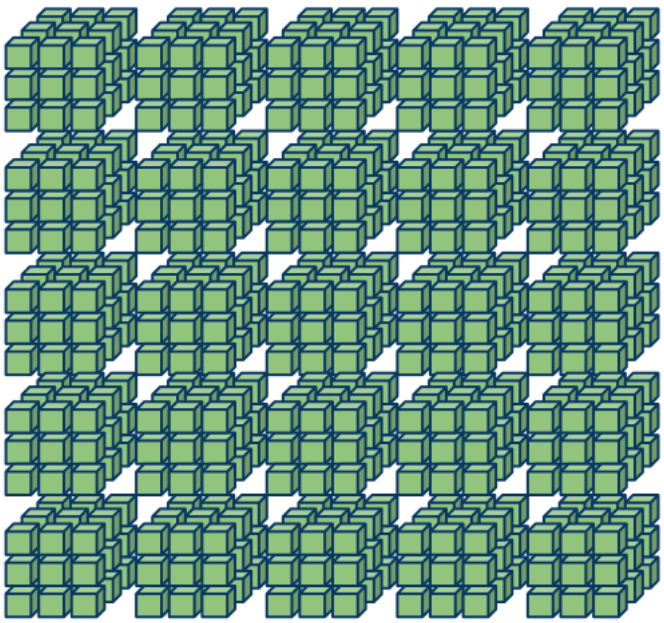

Figura 1.5: *GRID* (B 5x5, T 3x3x3).

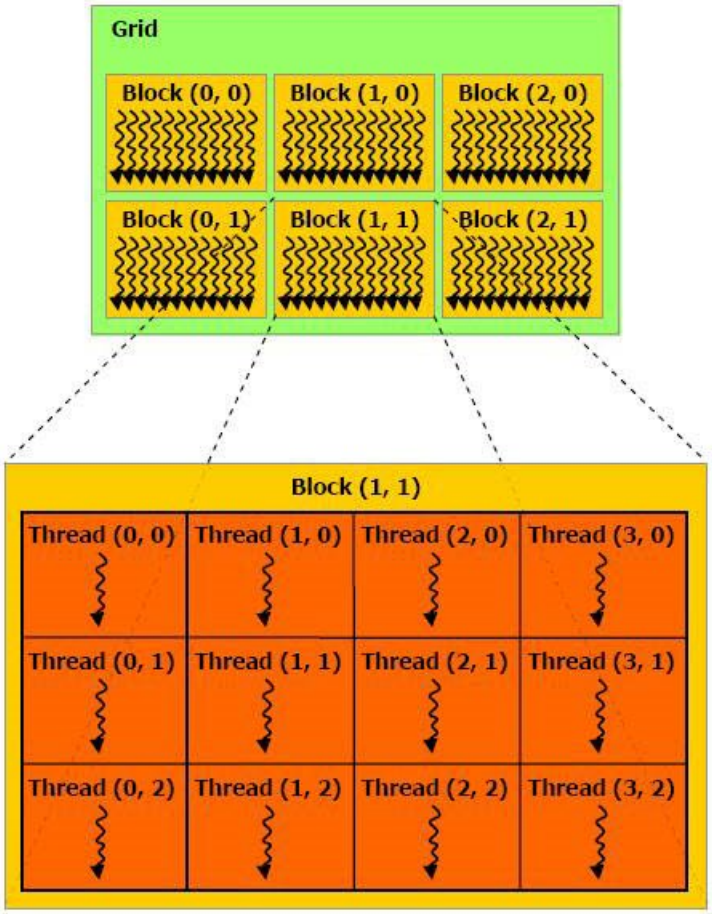

Figura 1.6: Índice de *Threads* no bloco.

conforme é mostrado na figura 1.7.

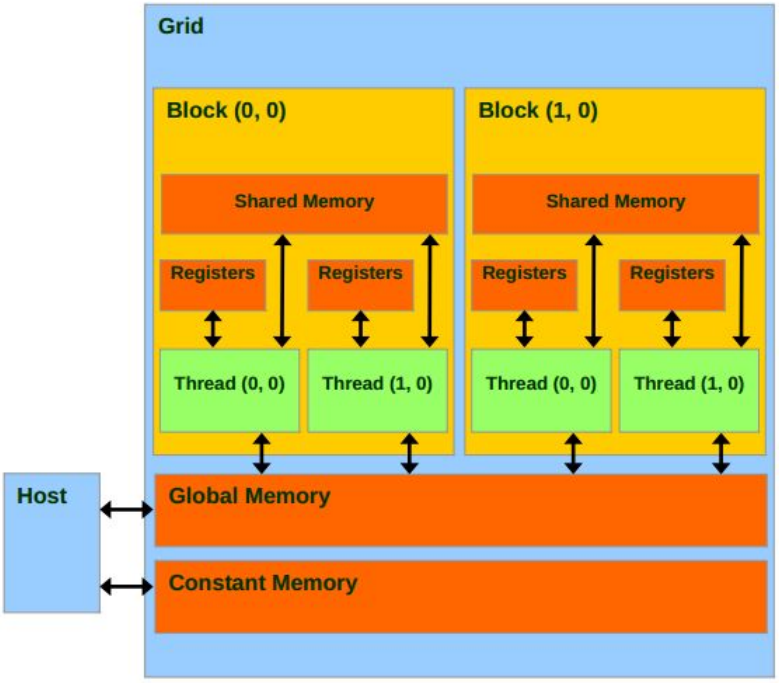

Figura 1.7: Modelo de *Grid* e Blocos da *GPU*.

Vejamos agora como é a sequência de execução de um programa que usa a tecnologia CUDA, conforme é mostrado na figura 1.8.

Cada função é executada em uma única *thread*. Para cada *thread* ser responsável por uma única execução, a solução é cada *thread* ter um único identificador [5].

Esse número de identificação é calculado usado o tamanho do *grid* e o tamanho de cada bloco, conforme é mostrado nas figuras 1.9 e 1.10.

No modelo em questão temos um *grid* com 2D blocos (3x3) que por sua vez possui 2D *threads* (3x3). Para o cálculo de cada índice das *threads* é feito da seguinte forma.

Índice do bloco = blockIdx.x + blockIdx.y * gridDim.x; Índice da thread = blockId * (blockDim.x * blockDim.y) + (threadIdx.y * blockDim.x) + threadIdx.x;

Lembrando que podemos ter até 3 dimensões nos *grids* e blocos, conforme é mostrado na figura 1.11.

Para se escrever um programa para *GPU*, tipicamente escreve-se um programa em C/C++, que chama uma ou mais funções escritas em CU. Essas funções são chamadas de *kernel*. O programa em C/C++ *(host)* passa dados para a *GPU*, que executa o *kernel (device)*, e retorna o valor processado para o programa em C/C++.

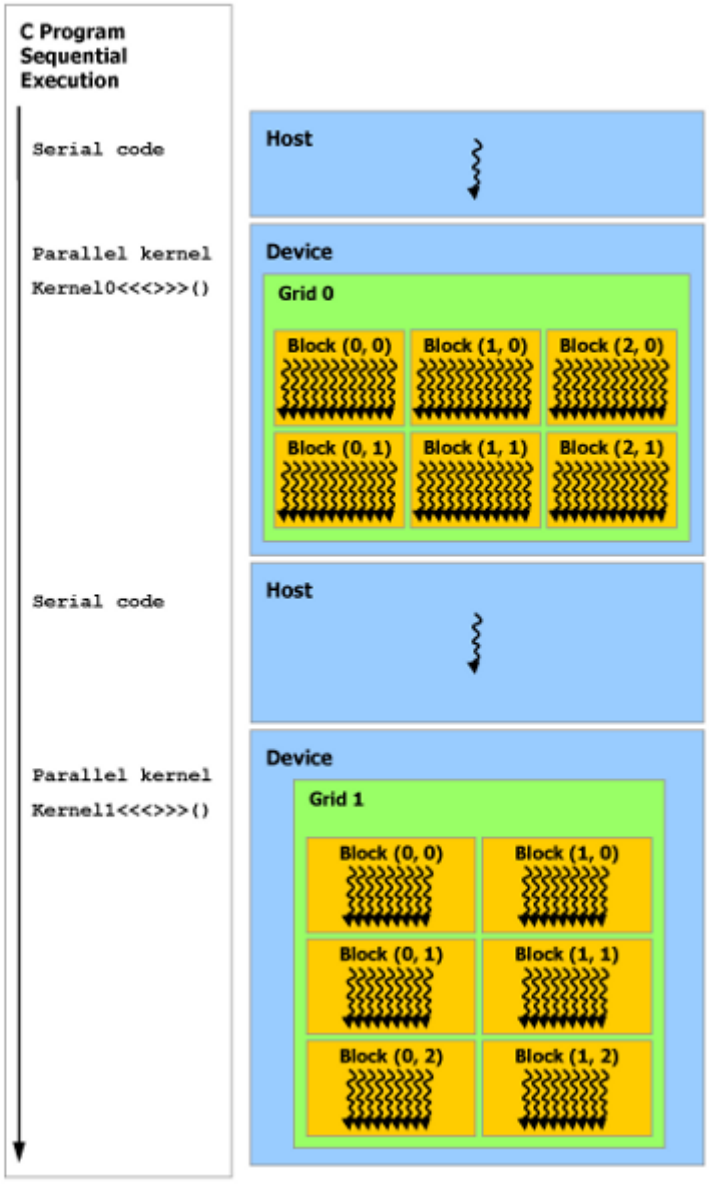

Figura 1.8: Sequência de execução CUDA.

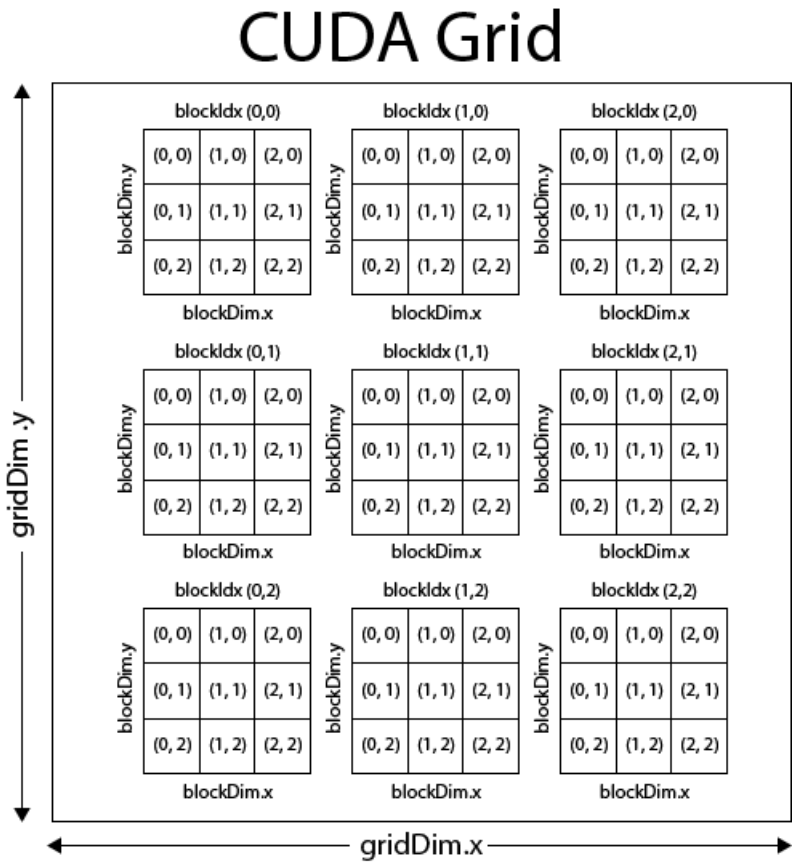

Figura 1.9: Modelo de *grid* para cálculo.

O código que é executado dentro da *GPU* é chamado através de uma função no estilo de programação C que chamamos de *kernel* [22]. Precisa ser definido como uma função que o código no *host* irá chamar e possui toda uma sintaxe diferenciada para ser chamado e escrito, possui muita similaridade com uma função comum em código C, conforme é mostrado na figura 1.12.

1.5 CUDA na prática

A plataforma CUDA possui algumas regras para sua execução. Uma delas é que a memória dentro da *GPU* não pode ser acessada diretamente. É necessário que seja alocado memória dentro da *GPU*. Após essa alocação ser bem sucedida é necessário que a memória principal *(host)* seja copiada para dentro da memória da *GPU (device)*, previamente alocada. Após essa operação de transferência de conteúdo de memória ter acontecido é que podemos usar o processamento paralelo dentro da *GPU*. Uma função escrita de maneira específica usando a sintaxe da plataforma

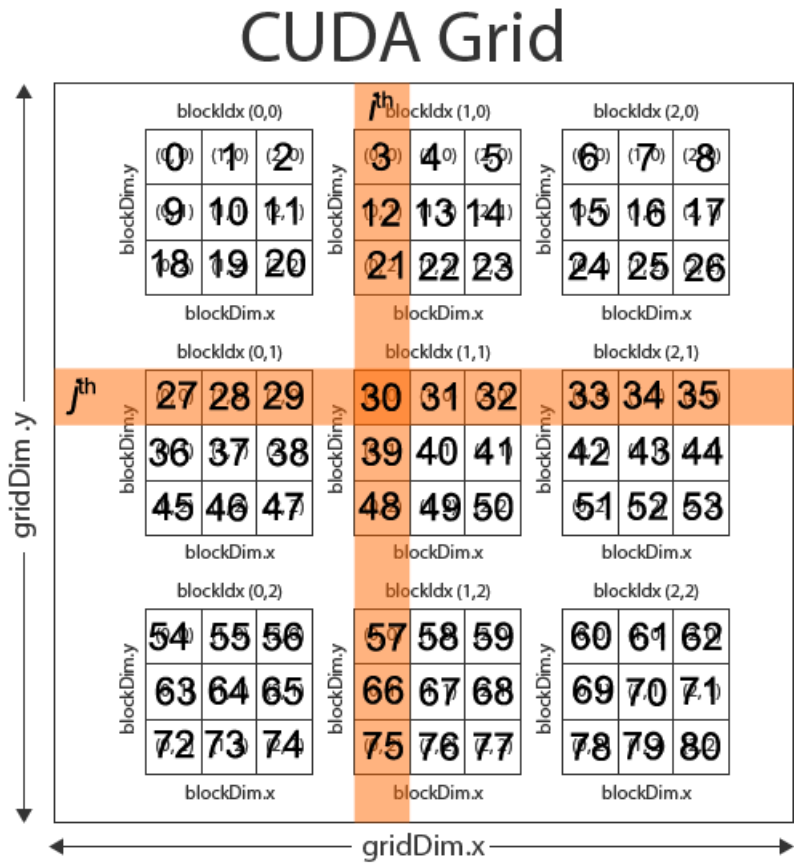

Figura 1.10: Índice de cada *thread*.

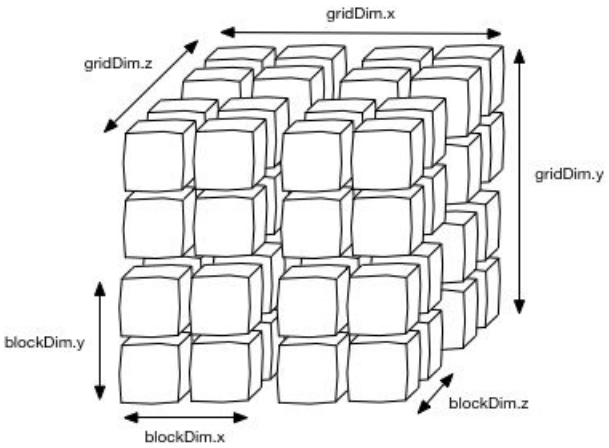

Figura 1.11: *Grid* tridimensional.

```
#include <iostream>

__global__ void kernel( void ) {
}

int main( void ) {
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

Figura 1.12: Função void *kernel(void)*.

CUDA será invocada e manipulará os dados copiados para a memória da *GPU*. Logo em seguida o processador deixa o processamento por conta da *GPU* que utiliza os seus núcleos CUDA que são chamados de *cuda-cores* para fazer o processamento em paralelo. Após o término desse processamento na *GPU* é preciso que a memória onde estão os resultados seja copiada de volta para a memória principal, ou seja, o caminho inverso é feito.

Os passos para a alocação de memória, processamento e retorno do resultado são mostrados no algoritmo em 1.13 e descritos abaixo:

Linhas 17 e 18, alocação de memória na *GPU*.

Linha 20, cópia da memória principal para a memória da *GPU*.

Linha 27, chamada ao *kernel* vecChange(gridSize, blockSize).

Linha 29 o resultado do processamento é copiado de volta para a memória principal.

Linhas 31 e 32, a memória do dispositivo é liberada.

Como parâmetro do *kernel* (função executada na *GPU*) é necessário passar qual o tamanho do *grid* (quantidade de blocos) e o tamanho do bloco (quantidade de *threads*). Esses parâmetros são passados nas variáveis *gridSize* e *blockSize*.

1.6 Biblioteca BLAS

A biblioteca BLAS *(Basic Linear Algebra Subprograms)* foi publicada pela primeira vez em 1979 sendo desenvolvida em FORTRAN e fortemente utilizada até hoje.

É amplamente usada em *softwares* de engenharia e pacotes matemáticos. De-

```
__global__ void vecChange(double *a, double *c, int n)
{   // CUDA kernel. Each thread takes care of one element of c
    int id = blockIdx.x*blockDim.x+threadIdx.x;   // Get our global thread ID
    if (id < n)    // Make sure we do not go out of bounds
        c[id] = a[id];
}
int main( int argc, char* argv[] )
{
    int n = 100000;
    double *h_a, *h_c, ;    // Host input vectors
    double *d_a, *d_c;      // Device input vectors
    size_t bytes = n*sizeof(double); // Size, in bytes, of each vector
    // Allocate memory for each vector on host
    h_a = (double*)malloc(bytes);
    h_c = (double*)malloc(bytes);
    // Allocate memory for each vector on GPU
    cudaMalloc(&d_a, bytes);
    cudaMalloc(&d_c, bytes);
    // Copy host vectors to device
    cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
    int blockSize, gridSize;
    // Number of threads in each thread block
    blockSize = 1024;
    // Number of thread blocks in grid
    gridSize = (int)ceil((float)n/blockSize);
    // Execute the kernel
    vecChange<<<gridSize, blockSize>>>(d_a, d_c, n);
    // Copy array back to host
    cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );
    // Release device memory
    cudaFree(d_a);
    cudaFree(d_c);
    // Release host memory
    free(h_a);
    free(h_c);
    return 0;
}
```

Figura 1.13: Alocação de memória no dispositivo.

vido a entrada da computação na programação numérica várias operações foram inseridas dentro da biblioteca para uso comum. Obteve grande confiabilidade pelos desenvolvedores devido ao intenso uso na indústria e academia. É sem dúvida um padrão para operações computacionais de álgebra linear. Diversos fabricantes de *hardware* otimizaram a implementação em sua própria arquitetura devido a todo esse reconhecimento do mercado. São eles:

- AMD [1]
- Apple [2]
- Compaq [4]
- Cray [6]
- HP [8]
- IBM [9]
- Intel [10]
- NEC [13]
- SGI [23]
- SUN [21]

O uso dessa biblioteca se divide em funções de primeiro, segundo e terceiro nível. São funções de primeiro nível operações com vetores.

$$y \leftarrow \alpha x + y \tag{1.1}$$

São funções de segundo nível operações de vetores com matrizes.

$$y \leftarrow \alpha Ax + \beta y \tag{1.2}$$

São funções de terceiro nível operações matriciais.

$$C \leftarrow \alpha AB + \beta C \tag{1.3}$$

1.7 CBLAS

A biblioteca BLAS desenvolvida em FORTRAN ganhou notoriedade e logo foi desenvolvida uma camada de interface com a linguagem C obtendo o nome de CBLAS [14].

1.8 Biblioteca cuBLAS

A biblioteca cuBLAS é a implementação da biblioteca BLAS, utilizando o paralelismo oferecido pela tecnologia CUDA. Todas as 152 rotinas da biblioteca BLAS foram aceleradas usando a tecnologia CUDA.

Como falamos anteriormente devido ao forte uso e confiabilidade que existem nas rotinas da biblioteca BLAS que a biblioteca cuBLAS foi construída.

A biblioteca cuBLAS funciona de maneira parecida com a BLAS, possui funções de primeiro, segundo e terceiro nível, respeitando o mesmo funcionamento da biblioteca BLAS, para melhor entendimento.

1.9 Vantagens do uso de cuBLAS

Com uso da biblioteca cuBLAS podemos utilizar todo o paralelismo existente na tecnologia CUDA. A biblioteca cuBlas foi construída em cima da mais famosa biblioteca de álgebra linear já desenvolvida que é a BLAS. A biblioteca BLAS é uma referência em álgebra linear. Podendo ser considerada como a principal biblioteca para resolução de problemas em álgebra linear.

Ao longo de várias décadas, desenvolvedores tanto no ambiente acadêmico, como no profissional, vem utilizando as suas rotinas.

Todo esse histórico da biblioteca BLAS nos deixa bastante confiante no uso da cuBLAS. O uso da biblioteca cuBLAS é 100% feito a partir da linguagem C/C++, sem uso de linguagem CU diretamente. Isso faz o desenvolvimento de software abstrair-se completamente do complexo código CU, o que torna tudo muito mais manutenível.

No código CUDA nativo é preciso estar atento a vários detalhes. Toda alocação e gerência de memória é de responsabilidade do desenvolvedor CUDA. O cálculo dos índices das *threads* é de responsabilidade do desenvolvedor. É também necessário

gerenciar todos os quatro níveis de memória existentes na *GPU*. Todo o cálculo do tamanho do *grid* e do bloco a ser executadas é de responsabilidade do desenvolvedor. Todo esse processo, no desenvolvimento de CUDA nativo é feito por conta do desenvolvedor, deixando assim a produtividade com baixos níveis de rendimento e forçando o desenvolvimento de exaustivos testes no código nativo CUDA.

É importante salientar que a curva de aprendizado é lenta pois para gerar resultados satisfatórios é preciso muito tempo de dedicação no aprendizado da tecnologia.

O resultado final de uma implementação de um código CUDA contém um alto grau de sofisticação. Para se fazer qualquer alteração é demandado um enorme esforço de manutenção acarretando um alto investimento de tempo. Isso resulta em um código com baixa manutenibilidade.

Ainda podemos citar que para se obter resultados satisfatórios são necessários vários testes visando a sintonia fina de todo esse conjunto de regras que a plataforma nos impõe.

A cuBLAS implementa de forma paralela todas as funções da biblioteca BLAS nos deixando completamente despreocupados de muitos detalhes inerentes a tecnologia CUDA.

É uma grande vantagem de não se ter de preocupar com a difícil sintaxe de CUDA. Todos a regras de implementação da plataforma são deixadas de lado. Ficamos totalmente focados somente em analisar os pontos paralelizáveis em nosso código e aplicar a biblioteca diretamente nesses pontos. Deixando nosso código com alta manutenibilidade e fazendo a construção do software de forma compacta. Todos os ajustes de invocação do *kernel* CUDA é feito pela biblioteca. Estamos deixando todos esses detalhes dentro de uma biblioteca que já foi exaustivamente testada para fazer esse trabalho por nós. A aplicação da biblioteca nos deixa livre de uma série de problemas até se obter um código realmente eficiente.

Isso nos dá de longe uma alta produtividade, pois só estamos manipulando funções que irão acessar toda a parte complicada da arquitetura CUDA.

Dessa forma estamos lidando com uma camada mais alta de desenvolvimento, podemos dizer que estamos trocando um pouco de performance em se escrever o código nativo em CUDA pelo alto nível de manutenibilidade e confiabilidade em se usar uma biblioteca. Essa forma também respeita o padrão de *software* que prega

o reuso em formas de biblioteca, deixando o código mais legível e mais robusto.

Além desses inúmeros benefícios, é um ponto importante que ao se usar uma biblioteca estamos deixando o código com baixo grau de acoplamento, nos dando a liberdade de fazer testes com outras bibliotecas disponíveis no mercado. É como a engenharia de *software* recomenda a construção de um software com partes independentes e isoladas se comunicando através de fluxos conhecidos e bem definidos [3].

Com o uso da biblioteca podemos mensurar exatamente de forma simples em quais pontos estamos tendo resultados significativos com baixa manutenção no código.

A ciência da engenharia de *software* nos ensina que cada vez mais que deixamos o nosso código em blocos, mais estamos cooperando para o reuso e manutenção do mesmo.

Esses são argumentos sólidos na hora de construirmos um software de qualidade. Foi com a análise de todos esses fatores que tomamos a decisão de usar a biblioteca cuBLAS.

Capítulo 2

Otimização Irrestrita com L-BFGS

2.1 Breve descrição do Algoritmo BFGS

Dentro do ramo de otimização o algoritmo BFGS Broyden-Fletcher-Goldfarb-Shanno é uma aproximação do método de Newton sendo um algoritmo interativo para resolução de problemas de otimização não-lineares sem restrição. A condição de otimalidade de primeira ordem é que o gradiente seja zero. A condição de otimalidade de segunda ordem, que minimiza a função de custo, é que a Hessiana seja maior que zero. O método BFGS usa o gradiente da função e estima a Hessiana, de modo a levar em conta a curvatura da função de custo para a determinação do caminho de minimização.

No método BFGS é guardada toda a matriz Hessiana que pode ser muito custoso e não suporta problemas com um grande número de variáveis. Como exemplo, seja um problema em $\mathbb{R}^n$ com $n = 1000$. A matriz Hessiana terá nesse caso dimensão 1000×1000.

2.2 Breve Descrição do Algoritmo-BFGS

O método L-BFGS [16] é um algoritmo, com código fonte disponível em [20], de otimização na família de métodos quase-Newton que se aproxima o algoritmo Broyden-Fletcher-Goldfarb-Shanno (BFGS) usando uma quantidade limitada de memória do computador. Tornando viável a solução de problemas com um número maior de variáveis.

Esse método foi proposto por Jorge Nocedal, e o código fonte em C usado nessa tese foi desenvolvido por seu orientando Naoaki Okazaki.

2.3 Conceito de Otimização

A Otimização é uma ciência que vem se desenvolvendo como uma forte ferramenta para decisão de sistemas físicos. A primeira abordagem do seu uso é ter um objetivo bem definido e uma medida que mostre o desempenho do objeto de estudo, o sistema. A primeira parte de identificação, como variáveis, restrições e objetivo é chamado de modelagem. Após essa modelagem do modelo em questão, uma técnica pode ser usada para encontrar as suas soluções viáveis e até mesmo a solução ótima.

Não existe uma solução geral sempre aplicável para problemas de otimização. Existem várias técnicas e métodos que atuam melhor sobre uma modelagem de um tipo de problema de otimização. A escolha da técnica a ser aplicada no modelo é de particular decisão do analista que está sobre o caso. Existem algoritmos específicos para atacarem certos modelos que já se mostraram úteis em outras experiências, parte do analista ter o conhecimento necessário para aplicar a melhor técnica.

Com uma técnica ou algoritmo sendo aplicado a um problema de otimização deve ser levar em conta o tempo e o uso de recurso que foram gastos na obtenção da solução. Esse resultado deve ser apurado com um conjunto de regras para se verificar se a condição de otimalidade foi satisfeita. O algoritmo ou técnica deve procurar soluções que satisfaçam as restrições e seja o resultado ótimo do problema.

Os algoritmos de otimização são classificados em grupos. Um exemplo importante desse tipo de grupo é o de algoritmo de otimização interativo. Os algoritmos desse grupo partem de um ponto inicial e seguem um caminho buscando pontos que reduzam o valor da função de custo, e que satisfaçam as restrições se houver. Um critério de parada existe e permite que o algoritmo possa terminar.

Abaixo estão listadas algumas propriedades esperadas dos algoritmos de otimização.

- Precisão: o algoritmo de otimização deve ser capaz de produzir uma solução próxima do ótimo do problema.

- Eficiência: o algoritmo de otimização deve ser capaz de resolver problemas complexos a partir da alocação de poucos recursos computacionais.

- Robustez: o algoritmo deve retornar a solução ótima rapidamente para uma ampla gama de valores dos seus parâmetros de entrada; notadamente o ponto inicial.

O objetivo desse trabalho é o estudo do algoritmo L-BFGS que é de otimização não-linear irrestrita, cujo objetivo é determinar o valor do argumento que minimiza uma função de custo que seja dependente de variáveis reais, sem restrições nos valores dessas variáveis.

Se uma função objetivo f é duas vezes continuamente diferenciável, pode-se de dizer que x^* é um ponto de mínimo local, se $\nabla f(x^*) = 0$ e a Hessiana $\nabla^2 f(x^*) \geq 0$(Hessiana semidefinida possitiva).

Segundo o Teorema de Taylor: Seja $f : \mathbb{R}^n \to \mathbb{R}$ continuamente diferenciável. Seja $p, x \in \mathbb{R}^n$. Logo

$$f(x+p) = f(x) + \nabla f(x+tp)^T p \tag{2.1}$$

para algum $t \in (0,1)$.

Ainda assim, se f é duas vezes continuamente diferenciável, então

$$f(x+p) = f(x) + \nabla f(x)^T p + \frac{1}{2} p^T \nabla^2 f(x+tp)^T p \tag{2.2}$$

para algum $t \in (0,1)$.

Inúmeros algoritmos de minimização exigem um ponto inicial, que no caso geral é definido por x^0. Tendo o ponto x^0 como inicial os algoritmos passam por k iterações e finalizam quando o critério de parada estabelecido é encontrado. Podemos assumir um critério de parada quando o valor da iteração x^k e x^{k-1} for menor que um determinado limite.

Ao decidir como passar da iteração k para a próxima, os algoritmos usam informações sobre o valor da função f em x^k e possivelmente informações de iterações anteriores $x^{k-1}, \ldots, x^1, x^0$, e também ∇f e $\nabla^2 f$. Estas informações são usadas para encontrar uma nova iteração $f(x^{k+1}) < f(x^k)$.

Há duas estratégias fundamentais para passar do ponto x^k atual para um novo ponto x^{k+1} da próxima iteração. O algoritmo escolhe a direção de busca p_k e também o passo de busca α de modo a determinar o x da próxima iteração:

$$x^{k+1} = x^k + \alpha p_k \tag{2.3}$$

O valor do passo α é determinado resolvendo-se o problema de minimização unidimensional da equação 2.4.

$$\alpha = arg\ min\ [f(x_k + \alpha p_k)], \quad \alpha > 0 \tag{2.4}$$

Encontrando-se a solução do problema 2.4, determina-se o valor do x da próxima equação que produz a máxima minimização da função de custo. A minimização exata é computacionalmente cara e desnecessária. Os algoritmos de otimização irrestrita geralmente resolvem o problema 2.4 sub-ótimo rapidamente. O novo ponto é determinado pela aplicação do valor apurado pela equação 2.4 sobre a equação 2.3.

Um critério de parada, conhecida como *Regra de Armijo* é o que estipula que α deve primeiramente fornecer uma redução suficiente na função objetivo f da seguinte forma da equação 2.5, sendo c_1 e c_2 parâmetros do critério de parada.

$$f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla f_k^T p_k,\ 0 < c_1 < 1 \tag{2.5}$$

Condição de parada para busca de linha inexata são as chamadas *Condições Wolfe*:

$$\begin{aligned} & f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla f_k^T p_k \\ & \nabla f(x_k + \alpha p_k)^T p_k \geq c_2 \nabla f_k^T p_k \\ & 0 < c_1 < c_2 < 1 \end{aligned} \tag{2.6}$$

Na regra de Wolfe, além da imposição de uma redução no valor da função objetivo, também é imposta uma restrição ao novo valor do gradiente da função no ponto $x_k + \alpha_k p_k$.

Em outra estratégia algorítmica, conhecida como *Região de Confiança*, as informações adquiridas sobre a função objetivo f são usadas para construir um modelo

m_k cujo comportamento próximo do ponto atual x_k é semelhante ao comportamento real de f.

Como o modelo m_k pode não ser uma boa aproximação de f quando $x*$ está longe de x_k, a ideia é restringir a procura por um ponto de mínimo local de m_k para alguma região em torno de x_k. Em outras palavras, encontra-se o passo de minimização candidato p resolvendo-se o subproblema :

$$minimizar\ p\ em\ m_k(x_k + p) \tag{2.7}$$

onde $x_k + p$ está dentro da região de confiança.

Se a solução encontrada não produzir uma redução suficiente no valor da função f, pode-se concluir que a região de confiança é muito grande, reduzi-la e voltar a resolver o problema.

O modelo m_k é geralmente definido como uma função quadrática da forma

$$m_k(x_k + p) = f_k + p^T \nabla f_k + \frac{1}{2} p^T B_k p \tag{2.8}$$

onde f_k, ∇f_k e B_k são um escalar, um vetor e uma matriz, respectivamente.

A matriz B_k é a Hessiana $\nabla^2 f_k$ ou alguma aproximação dessa matriz.

Em um certo sentido, as abordagens, busca em linha e região de confiança, diferem na ordem em que são escolhidas a direção e distância a ser percorrida para a próxima iteração. A busca em linha começa pela fixação de uma direção p_k para, em seguida, identificar uma distância adequada a ser percorrida, ou seja, o comprimento do passo α_k.

No método da região de confiança, primeiro é escolhido uma distância máxima (o raio da região de confiança k) e, em seguida, procura-se uma direção e um tamanho de passo que alcancem a maior redução possível no valor de f com esta restrição de distância. Se essa etapa revelar-se insatisfatória, o raio da região de confiança k é reduzido e uma nova tentativa de se encontrar um ponto que diminua o valor da função objetivo é feita.

Uma importante direção de busca é chamada de *Direção de Newton.* Essa direção é originada da aproximação da série de Taylor de segunda ordem para $f(x_k + p)$. Métodos que usam a direção de Newton têm uma taxa rápida de convergência local,

tipicamente quadrática.

Uma desvantagem do algoritmo baseado na direção de Newton é a necessidade da codificação (em linguagem de programação) de todos os elementos da matriz Hessiana $\nabla^2 f(x)$ no *callback* da função de custo. A computação explícita desta matriz de segundas derivadas é, na maioria das vezes, um processo caro e propenso a erros de arredondamento. Mesmo com máquinas muito velozes computacionalmente falando, calcular a Hessiana é um processo muito custoso.

Como alternativa, existem as chamadas direções *Quasi-Newton* de busca, as quais fornecem uma opção atraente na medida em que não requerem computação explícita da matriz Hessiana e ainda assim atingem uma taxa de convergência superlinear. No lugar da verdadeira matriz Hessiana $\nabla^2 f(x)$, é utilizada uma aproximação da mesma, usualmente definida como B_k, que é atualizada após cada iteração para se levar em conta o conhecimento adicional sobre a função objetivo e seu gradiente adquiridos durante a iteração atual.

As atualizações na matriz B_k fazem uso do fato de que as mudanças no gradiente fornecem informações sobre a segunda derivada da função objetivo ao longo da direção de busca. A nova aproximação para a Hessiana B_{k+1} é calculada para satisfazer a seguinte condição, conhecida como *equação da secante*:

$$\begin{aligned} &B_{k+1}s_k = y_k, \quad onde \\ &s_k = x_{k+1} - x_k \quad e \\ &y_k = \nabla f_{k+1} - \nabla f_k \end{aligned} \tag{2.9}$$

Tipicamente, requisitos adicionais são impostos em B_{k+1}, tais como simetria (motivada pela simetria da matriz Hessiana exata) e uma restrição onde a diferença entre aproximações sucessivas B_k e B_{k+1} tenha um baixo posto. A aproximação inicial B_0 deve ser escolhida pelo desenvolvedor do modelo, geralmente atribui-se a matriz I.

2.4 BFGS

Uma das fórmulas mais populares para atualizar a aproximação da matriz Hessiana B_k é a fórmula BFGS, nomeada a partir das iniciais de seus inventores, Broyden, Fletcher, Goldfarb e Shanno, que é definida por

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k} \tag{2.10}$$

A direção de busca quasi-Newton é dada por B_k substituindo a matriz Hessiana exata na fórmula:

$$p_k = -B_k^{-1} \nabla f_k \tag{2.11}$$

Algumas implementações práticas de métodos Quasi-Newton evitam a necessidade de se fatorar B_k a cada iteração atualizando a inversa da matriz B_k em vez da matriz B_k em si.

Após a atualização da matriz b_k, a nova iteração é dada por:

$$x_{k+1} = x_k + \alpha_k p_k \tag{2.12}$$

onde o comprimento do passo α_k é escolhido para satisfazer as condições de Wolfe, por exemplo.

A inversa da matriz B_k, chamada de H_k, é atualizada através da seguinte fórmula:

$$H_{k+1} = (I - \rho_k s_k y_k^T) H_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T \tag{2.13}$$

onde

$$\rho_k = \frac{1}{y_k^T s_k} \tag{2.14}$$

Uma versão simplificada do algoritmo BFGS pode ser definida como:

O método BFGS tem taxa de convergência superlinear.

Os métodos Quasi-Newton não são diretamente aplicáveis a grandes problemas de otimização porque as aproximações da matriz Hessiana ou a sua inversa são geralmente densas. Nestes casos, usam-se métodos Quasi-Newton com memória

Algoritmo 1 Algoritmo do método BFGS

Dado um ponto inicial x_0, uma tolerância $\epsilon > 0$ e uma aproximação para a matriz Hessiana H_0;

$k \leftarrow 0$
while $\|\nabla f_k\| > \epsilon$ **do**
  $p_k \leftarrow -H_k \nabla f_k$ {Calcular a direção de busca p_k}
  $x_{k+1} \leftarrow x_k + \alpha_k p_k$ {Onde α_k é calculado a partir de um procedimento de busca em linha que satisfaça as condições de Wolfe}
  $s_k \leftarrow x_{k+1} - x_k$
  $y_k \leftarrow \nabla f_{k+1} - \nabla f_k$
  Calcular H_{k+1} *usando* (2.12)
  $k \leftarrow k + 1$
end while

limitada. Estes métodos são úteis para resolver problemas de grande porte cuja matriz Hessiana não pode ser computada a um custo razoável ou é demasiadamente densa para ser manipulada facilmente.

2.5 L-BFGS

Nocedal em [16] propôs um método do tipo Quasi-Newton com memória limitada, o L-BFGS [20], que mantém aproximações simples de matrizes Hessianas, e ao invés de armazenar plenamente essa matriz de aproximação densa, são armazenados apenas poucos vetores que representam as aproximações de forma implícita.

O L-BFGS muitas vezes consegue uma taxa de rendimento aceitável de convergência, normalmente linear. A estratégia de guardar os últimos m pares de vetores funciona bem na prática, apesar de não ser recomendável para problemas muito mal condicionados, em que os autovalores são distantes uns dos outros. Outra opção seria guardar apenas os pares de vetores que gerem matrizes bem condicionadas.

A ideia principal do método L-BFGS é usar informações de curvatura apenas a partir das iterações anteriores para construir a aproximação da matriz Hessiana.

Um dos principais pontos fracos do método L-BFGS é que ele muitas vezes converge lentamente, o que geralmente leva a um número relativamente grande de avaliações da função objetivo. Além disso, o método é altamente ineficiente em problemas mal condicionados, especificamente nos casos onde a matriz Hessiana contém uma ampla distribuição de autovalores. Existem implementações livres do algoritmo BFGS.

Vale lembrar que Nocedal ganhou o prêmio Dantzig em 2012 pela formulação do algoritmo L-BFGS. Com o advento de se guardar poucos vetores da matriz Hessiana tornou-se possível a aplicação do método em problemas com muitas variáveis.

Capítulo 3

Paralelização do Método L-BFGS com Tecnologia CUDA

3.1 Função de Custo

A função de Rosenbrock [12], mostrada em 3.1, é uma função clássica de teste na ciência da otimização. Muitas vezes referenciada como a função banana de Rosenbrock devido a forma das suas curvas de níveis, conforme mostrado em 3.1. A função é unimodal, e o mínimo global situa-se num vale parabólico estreito. No entanto, mesmo que este vale seja fácil de encontrar, a convergência para o mínimo global é difícil de se achar.

$$f(x) = \sum_{i=1}^{n-1} 100\,(x_{i+1} - x_i^2)^2 + (x_i - 1)^2, \;\; x \in \mathbb{R}^n \tag{3.1}$$

Devido ao seu alto grau de dificuldade em se achar o mínimo global, precisando assim de várias iterações do algoritmo, foi que a função de Rosenbrock foi escolhida para execução dos nossos teste.

3.2 Implementação do Paralelismo

No processo de implementação paralela do método L-BFGS utilizaremos a biblioteca cuBLAS. Ao fazermos o uso de uma biblioteca de funções estamos deixando o código fonte bem menos acoplado. Essa é uma técnica muito recomendada na engenharia de software. Estamos deixando o código fonte com baixo acoplamento, mais elegante

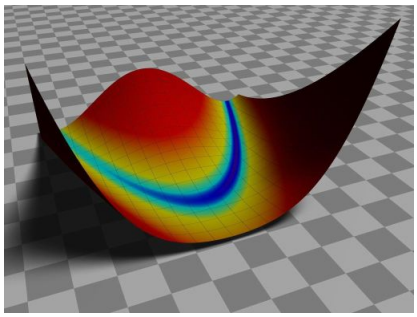

Figura 3.1: Função de Rosenbrock, para $x \in \mathbb{R}^2$.

e com grande manutenibilidade. Esses são fatores fundamentais para se construir um software com alta qualidade e robustez [3].

O método L-BFGS [20] possui grande parte de sua construção em cima de operações vetoriais. Operações com vetores são efetuadas massivamente durante todo o processo. Para um rápido exemplo a obtenção da aproximação da inversa da Hessiana como mostrado em 2.10.

Chamando-se as funções da biblioteca cuBLAS para a execução dessas operações vetoriais faz-se o uso de *GPU*. Como será visto nos resultados, isso afeta o desempenho e em alguns caso produz *speedup*. Foi identificado no código fonte do L-BFGS vários pontos onde chamou-se funções da biblioteca cuBLAS para execução de operações vetoriais.

3.3 Biblioteca cuBLAS

Foi feito um estudo detalhado da biblioteca cuBLAS; esse estudo consiste na forma adequada do seu funcionamento e aplicação. Como dito anteriormente a biblioteca deriva suas funções da biblioteca BLAS. A biblioteca cuBLAS possui algumas particularidades relevantes. Ela é aplicada a somente uma *GPU*, ou seja, o código só é enviado a uma *GPU*. Caso o computador possua mais de uma *GPU*, somente uma será utilizada. Existe a biblioteca cuBLAS-XT [18], que possui a tecnologia de enviar para mais de uma *GPU*. A biblioteca cuBLAS-XT reúne somente funções de terceiro nível, isto é, operações matriciais como descrito em 1.3. No caso da paralelização da biblioteca L-BFGS constatou-se que as operações paralelizáveis são apenas as BLAS de nível 1, conforme descrito em 1.1, ou seja, uma máquina com mais de uma *GPU* só utilizará uma delas no caso de executar o *software* L-BFGS

descrito neste trabalho.

3.4 Aspectos do Paralelismo com CUDA

Para o desenvolvimento de código CUDA nativo é necessário que se faça a alocação de memória no dispositivo como mostrado no fragmento de código 1.13, nas linhas 17 e 18. Na linha 20 copia-se os dados da memória do *host* (*cpu*) para o *device* (*gpu*).

O processamento consiste em mover dados do *host* (cpu) para o *device* (*gpu*), processar os dados no *device* e em seguida copiar os dados de volta para o *host*, conforme é mostrado na linha 29.

A comparação de *speedup* é feita da seguinte maneira: mede-se o desempenho da execução com a *cpu* e compara-se o resultado com o desempenho do conjunto de operações que consiste de:

1. Cópia para o *device*,
2. Processamento no *device*,
3. Cópia para o *host*.

3.5 Trechos Paralelizáveis do Algoritmo

O pacote original do L-BFGS consiste nos seguintes arquivos:

- arithmetic_ansi.h
- arithmetic_sse_double.h
- arithmetic_sse_float.h
- lbfgs.c
- lbfgs.h

No arquivo arithmetic_ansi.h estão declarados os protótipos das funções vetoriais usadas no algoritmo, e também as suas definições implementadas em C padrão.

No arquivo arithmetic_sse_double.h estão escritos a implementação das funções vetoriais com o uso das instruções especiais de CPU do tipo SSE double.

No arquivo arithmetic_sse_float.h estão escritos a implementação das funções vetoriais com o uso das instruções especiais de CPU do tipo SSE float.

No arquivo lbfgs.h estão escritos os protótipos das funções do algoritmo l-bfgs. No arquivo lbfgs.c estão implementadas as funções cujo os protótipos estão escritos no arquivo anterior.

Conforme está mostrado acima, o código original do L-BFGS foi desenvolvido por [20] com o conveniente cuidado de escrever o código com desacoplamento das funções vetoriais e outras operações matemáticas. O motivo original de ter sido feito assim, além da boa manutenabilidade do código, é permitir a implementação de aceleração de funções vetoriais a partir de códigos específicos de fabricantes de *cpu*.

O presente trabalho foi dividido em dois blocos:

1. Paralelização na camada de isolamento: Tomando proveito da arquitetura do L-BFGS com o isolamento das funções vetoriais declaradas no arquivo arithmetic_ansi.h, introduziu-se conexão com a biblioteca cuBLAS nesses pontos.

2. Paralelização fora da camada de isolamento: Analisando-se o corpo do algoritmo no arquivo lbfgs.c, foram identificadas trechos de chamadas vetoriais que foram substituídas com chamadas de funções da biblioteca cuBLAS com funcionalidade equivalente.

3.6 Bloco I: Paralelização na Camada de Isolamento

Alterou-se o arquivo arithmetic_ansi.h de forma a colocar as funções cuBLAS no lugar das funções originais de tratamento vetorial, tomando proveito do isolamento das funções vetoriais.

Como já foi mencionado, o algoritmo requer apenas o uso das funções BLAS de nível 1 1.1 e portanto não faz sentido o uso da biblioteca CUBLAS-XT que traria melhoria de desempenho apenas no caso do uso de funções BLAS nível 3 1.3.

As funções do arquivo arithmetic_ansi.h que foram modificadas para a sua versão equivalente com o uso da biblioteca cuBLAS estão listadas abaixo.

- vecset()
- veccpy()
- vecncpy()
- vecadd()
- vecdiff()
- vecscale()
- vecmul()
- vecdot()

3.6.1 Função vecset

```
inline static void vecset(lbfgsfloatval_t *x,
  const lbfgsfloatval_t c, const int n)
{
  int i;
  for (i = 0;i < n;++i) {
    x[i] = c;
  }
}
```

Função sequencial executada na CPU.

```
inline static void vecset(lbfgsfloatval_t *x,
  const lbfgsfloatval_t c, const int n)
{
  cuBLASSetVector(n, sizeof(lbfgsfloatval_t), &c, 1, d_x, 1);
}
```

Versão paralela usando a biblioteca cuBLAS.

3.6.2 Função veccpy

```
inline static void veccpy(lbfgsfloatval_t *y,
  const lbfgsfloatval_t *x, const int n)
{
  int i;
  for (i = 0;i < n;++i) {
    y[i] = x[i];
  }

}
```

Função sequencial executada na CPU.

```
inline static void veccpy(lbfgsfloatval_t *y,
  const lbfgsfloatval_t *x, const int n)
{
  cuBLASSetVector(n, sizeof(lbfgsfloatval_t), x, 1, d_x, 1);
  cuBLASDcopy(handle, n, d_x, 1, d_y, 1);
  cuBLASGetVector(n, sizeof(y[0]), d_y, 1, y, 1);
}
```

Versão paralela usando a biblioteca cuBLAS.

3.6.3 Função vecncpy

```
inline static void vecncpy(lbfgsfloatval_t *y,
  const lbfgsfloatval_t *x, const int n)
{
  int i;
  for (i = 0;i < n;++i) {
      y[i] = -x[i];
    }
  }
}
```

Função sequencial executada na CPU.

```
inline static void vecncpy(lbfgsfloatval_t *y,
  const lbfgsfloatval_t *x, const int n)
{
  cudaMemset(d_y, 0, n*sizeof(lbfgsfloatval_t));
  cuBLASSetVector(n, sizeof(lbfgsfloatval_t), x, 1, d_x, 1);
```

```
  cuBLASDaxpy(handle, n, &alpha, d_x, 1, d_y, 1);
  cuBLASGetVector(n, sizeof(y[0]), d_y, 1, y, 1);
}
```

Versão paralela usando a biblioteca cuBLAS.

3.6.4 Função vecadd

```
inline static void vecadd(lbfgsfloatval_t *y,
  const lbfgsfloatval_t *x, const lbfgsfloatval_t c, const int n)
{
  int i;
  for (i = 0;i < n;++i) {
    y[i] += c * x[i];
  }
}
```

Função sequencial executada na CPU.

```
inline static void vecadd(lbfgsfloatval_t *y,
  const lbfgsfloatval_t *x, const lbfgsfloatval_t c, const int n)
{
    cuBLASSetVector(n, sizeof(lbfgsfloatval_t), x, 1, d_x, 1);
    cuBLASSetVector(n, sizeof(lbfgsfloatval_t), y, 1, d_y, 1);
    cuBLASDaxpy(handle, n, &c, d_x, 1, d_y, 1);
    cuBLASGetVector(n, sizeof(y[0]), d_y, 1, y, 1);
}
```

Versão paralela usando a biblioteca cuBLAS.

3.6.5 Função vecdiff

```
inline static void vecdiff(lbfgsfloatval_t *z,
  const lbfgsfloatval_t *x, const lbfgsfloatval_t *y, const int n)
{
  int i;
  for (i = 0;i < n;++i) {
    z[i] = x[i] - y[i];
  }
}
```

Função sequencial executada na CPU.

```
inline static void vecdiff(lbfgsfloatval_t *z,
  const lbfgsfloatval_t *x, const lbfgsfloatval_t *y, const int n)
{
  cuBLASSetVector(n, sizeof(lbfgsfloatval_t), x, 1, d_x, 1);
  cuBLASSetVector(n, sizeof(lbfgsfloatval_t), y, 1, d_y, 1);
  cuBLASDcopy(handle, n, d_x, 1, d_result, 1);
  cuBLASDaxpy(handle, n, &alpha, d_y, 1, d_result, 1);
  cuBLASGetVector(n, sizeof(y[0]), d_result, 1, z, 1);
}
```

Versão paralela usando a biblioteca cuBLAS.

3.6.6 Função vecscale

```
inline static void vecscale(lbfgsfloatval_t *y,
  const lbfgsfloatval_t c, const int n)
{
  int i;
  for (i = 0;i < n;++i) {
    y[i] *= c;
  }
}
```

Função sequencial executada na CPU.

```
inline static void vecscale(lbfgsfloatval_t *y,
  const lbfgsfloatval_t c, const int n)
{
  cuBLASSetVector(n, sizeof(lbfgsfloatval_t), y, 1, d_y, 1);
  cuBLASDscal(handle, n, &c, d_y, 1);
  cuBLASGetVector(n, sizeof(y[0]), d_y, 1, y, 1);
}
```

Versão paralela usando a biblioteca cuBLAS.

3.6.7 Função vecmul

```
inline static void vecmul(lbfgsfloatval_t *y,
  const lbfgsfloatval_t *x, const int n)
{
```

```
  int i;
  for (i = 0;i < n;++i) {
    y[i] *= x[i];
  }

}
```

Função sequencial executada na CPU.

```
inline static void vecmul(lbfgsfloatval_t *y,
  const lbfgsfloatval_t *x, const int n)
{
  cuBLASSetVector(n, sizeof(lbfgsfloatval_t), x, 1, d_x, 1);
  cuBLASSetVector(n, sizeof(lbfgsfloatval_t), y, 1, d_y, 1);
  cuBLASDdot(handle, n, d_x, 1, d_y, 1, y);
}
```

Versão paralela usando a biblioteca cuBLAS.

3.6.8 Função vecdot

```
inline static void vecdot(lbfgsfloatval_t* s,
  const lbfgsfloatval_t *x, const lbfgsfloatval_t *y, const int n)
{
  int i;
  *s = 0.;
  for (i = 0;i < n;++i) {
    *s += x[i] * y[i];
  }
}
```

Função sequencial executada na CPU.

```
inline static void vecdot(lbfgsfloatval_t* s,
  const lbfgsfloatval_t *x, const lbfgsfloatval_t *y, const int n)
{
  cuBLASSetVector(n, sizeof(lbfgsfloatval_t), x, 1, d_x, 1);
  cuBLASSetVector(n, sizeof(lbfgsfloatval_t), y, 1, d_y, 1);
  cuBLASDdot(handle, n, d_x, 1, d_y, 1, s);
}
```

Versão paralela usando a biblioteca cuBLAS.

3.7 Bloco II: Paralelização Fora da Camada de Isolamento

Com o conhecimento acumulado na tecnologia CUDA foi concebido um método geral para se abordar o problema de adaptar um código C, com intenção de torná-lo paralelizado com a biblioteca cuBLAS. O método consiste basicamente nos seguintes passos:

1. Identificação de trechos de código onde ocorrem o uso subsequente de vetores com operações vetoriais.

2. Desenvolvimento de uma função que seja capaz de substituir as funcionalidades dos trechos definidos acima. Essa função deve otimizar a transferência de dados entre *host* e *device*, isto é, executar a transferência dos dados menos vezes do que seria se fosse feito com chamadas independentes da cuBLAS.

3. Substituição dos tipos de trechos pela função equivalente.

Foi feita uma inspeção criteriosa no arquivo lbfgs.c e foram encontrados trechos de código onde foi aplicado o método proposto, conforme explicado nas subseções abaixo.

3.7.1 Cálculo da Hessiana

Na implementação do cálculo da Hessiana, descrito em 2.10, foi identificado o seguinte tipo de trecho de código fonte com uso subsequente de vetores com operações vetoriais:

```
vecdot(&ys, it->y, it->s, n);
vecdot(&yy, it->y, it->y, n);
```

Foi desenvolvido a seguinte função capaz de substituir as funcionalidades do trecho acima.

```
inline static void my_vecdot(lbfgsfloatval_t* s1,
  lbfgsfloatval_t* s2, const lbfgsfloatval_t *x,
```

```
    const lbfgsfloatval_t *y, const int n)
{
    cublasSetVector(n, sizeof(lbfgsfloatval_t), x, 1, d_x, 1);
    cublasSetVector(n, sizeof(lbfgsfloatval_t), y, 1, d_y, 1);

    cublasDdot(handle, n, d_x, 1, d_y, 1, s1);
    cublasDdot(handle, n, d_y, 1, d_y, 1, s2);
}
```

O trecho original chama duas vezes a função `vecdot()`, que foi definida na sua implementação CUDA, e está descrita na subseção 3.6.8. O trecho realiza as seguintes operações. Como se pode ver são quatro operações de alocação de dados e transferência.

- Operação de alocação de dados no *device* e transferência para o vetor x.
- Operação de alocação de dados no *device* e transferência para o vetor y.
- Operação de alocação de dados no *device* e transferência para o vetor y.
- Operação de alocação de dados no *device* e transferência para o vetor y.

A função equivalente realiza as seguintes operações. Como se pode ver são duas operações de alocação de dados e transferência.

- Operação de alocação de dados no *device* e transferência para o vetor x.
- Operação de alocação de dados no *device* e transferência para o vetor y.

3.7.2 Busca Linear

Na implementação de busca linear, descrito em 2.6, foi identificado o seguinte trecho de código fonte com uso subsequente de vetores com operações vetoriais:

```
    veccpy(x, xp, n);
    vecadd(x, s, *stp, n);
```

Foi desenvolvido a seguinte função capaz de substituir as funcionalidades do trecho acima.

```
inline static void my_vec_copy_Add(lbfgsfloatval_t *x,
  const lbfgsfloatval_t *xp, lbfgsfloatval_t *s,
  lbfgsfloatval_t stp, const int n)
{
    cublasSetVector(n, sizeof(lbfgsfloatval_t), xp, 1, d_result, 1);
    cublasSetVector(n, sizeof(lbfgsfloatval_t), s, 1, d_y, 1);

    cublasDaxpy(handle, n, &stp, d_y, 1, d_result, 1);
    cublasGetVector(n, sizeof(lbfgsfloatval_t), d_result, 1, x, 1);
}
```

O trecho original chama as funções `veccpy()` e `vecadd()`, que foram definidas na sua implementação CUDA, e estão descritas nas subseções 3.6.4 e 3.6.2. O trecho realiza as seguintes operações:

- Operação de alocação de dados no *device* e transferência para o vetor x.
- Operação de alocação de dados no *device* e transferência para o vetor x.
- Operação de alocação de dados no *device* e transferência para o vetor y.

A função equivalente realiza as seguintes operações::

- Operação de alocação de dados no *device* e transferência para o vetor x.
- Operação de alocação de dados no *device* e transferência para o vetor y.

Capítulo 4

Resultados Experimentais

4.1 Introdução

Para a medição dos resultados experimentais serão comparados os tempos de execução do algoritmo original sequencial com as duas versões que foram produzidas com a versão adaptada do algoritmo baseado na tecnologia CUDA e na biblioteca cuBLAS. As duas versões correspondem a:

1. Implementado apenas o bloco I, ou seja, somente as funções na camada de isolamento foram substituídas por seus equivalentes chamando cuBLAS.

2. Implementado o bloco I e II, ou seja, além do bloco I também foram implementadas as substituições de trechos de código com o uso subsequente de vetores com operações vetoriais.

Em todos os casos repetiu-se o trabalho de perfilamento e desempenho para um conjunto de valores de n de $\mathbb{R}^n$, como mostrado na equação 4.1.

$$n \in \{10^4,\ 10^5,\ 10^6,\ 10^7,\ 10^8\} \tag{4.1}$$

Em todos os casos a função de custo a otimizar é a função de Rosenbrock em $\mathbb{R}^n$ mostrado em 3.1. A função de custo é sempre executada em sequencial (cpu), sendo paralelizado exclusivamente o algoritmo L-BFGS ele próprio.

Todos os testes foram realizados na máquina apelidada de "coyote" cujo o código é SGI C1104G-RP5, com 24 gigas de memória, 24 *cores* de *CPU* e 2 placas TESLA NVIDIA, com 448 *cores* de *GPU* cada, como mostrado na figura 4.1.

Figura 4.1: Coyote.

4.2 Perfilamento do Software

Para uma observação mais abrangente e detalhada dos fenômenos de *speedup* que se deseja analisar, entendeu-se como adequado fazer o perfilamento (*profile*) do *software* em questão, na versão original. A análise de perfilamento mostra o número de vezes que funções são chamadas e o tempo de execução de cada uma delas.

4.2.1 Número de Chamadas de Função

O número de chamadas agrupado por função está descrito na tabela 4.1.

	N					
Funções	10^3	10^4	10^5	10^6	10^7	10^8
vecset	0	0	0	0	0	0
veccpy	132	127	123	118	125	127
vecncpy	40	38	37	35	37	37
vecadd	490	465	451	426	453	455
vecdiff	78	74	72	68	72	72
vecscale	39	37	36	34	36	36
vecmul	0	0	0	0	0	0
vecdot	691	656	637	602	639	641

Tabela 4.1: Número de chamadas por função.

Como se poder ver, estranhamente, as funções *vecset*() e *vecmul*(), não são usadas e por isso o valor zero de ocorrência e tempo.

4.2.2 Tempo Gasto por Função

O resultado da medição do tempo gasto em cada função do algoritmo, mostrado na tabela 4.2.

	N				
Funções	10^4	10^5	10^6	10^7	10^8
vecset	0	0	0	0	0
veccpy	$\simeq$0	40	510	4580	46070
vecncpy	10	10	180	1500	15010
vecadd	20	230	2240	22470	227990
vecdiff	$\simeq$0	50	390	3420	34780
vecscale	$\simeq$0	30	100	1430	14160
vecmul	0	0	0	0	0
vecdot	50	350	2810	31080	314370

Tabela 4.2: Tempo(ms) separado por função.

O resultado do tempo total de execução do algoritmo separado pelo agrupamento de funções e restante do algoritmo é apresentado em 4.3.

	N				
Tempo	10^4	10^5	10^6	10^7	10^8
Funções	80	710	6230	64480	652380
Restante	510	570	970	5040	47350
Total	590	1280	7200	69520	699730

Tabela 4.3: Tempo total(ms) agrupado por função.

O resultado percentual de cada função em relação ao algoritmo é mostrado em 4.4.

	N				
Partes do algoritmo	10^4	10^5	10^6	10^7	10^8
Funções	13%	55%	86%	92%	93%
Restante	87%	45%	14%	8%	7%

Tabela 4.4: Percentual total agrupado.

4.3 Desempenho com o Bloco I

O resultado do tempo de execução após a implementação da primeira fase é mostrado em 4.5.

O resultado de *speedup* por função é comparado com a versão sequencial (*cpu*) é mostrado em 4.6.

	N				
Funções	10^4	10^5	10^6	10^7	10^8
vecset	0	0	0	0	0
veccpy	$\simeq$0	30	390	4460	43740
vecncpy	$\simeq$0	30	150	1330	13180
vecadd	20	230	2160	22430	228150
vecdiff	10	50	330	3450	34880
vecscale	$\simeq$0	40	110	1220	12390
vecmul	0	0	0	0	0
vecdot	80	410	2060	21980	219770

Tabela 4.5: Tempo (ms) separado por função.

	N				
Funções	10^4	10^5	10^6	10^7	10^8
veccpy	$\simeq$0	1.33	1.30	1.02	1.05
vecncpy	$\simeq$0	0.33	1.20	1.12	1.13
vecadd	$\simeq$ 1	$\simeq$1	1.03	1.01	0.99
vecdiff	$\simeq$ 0	$\simeq$1	1.18	0.99	0.99
vecscale	$\simeq$0	0.75	0.90	1.17	1.14
vecdot	0.62	0.85	1.36	1.41	1.43

Tabela 4.6: *Speedup* por função.

Como podemos ver em 4.6 as funções $vecadd()$ e $vecdiff()$ têm sua melhor performance quando executadas pela *CPU*, ou seja, perderam performance quando tiveram sua versão paralela implementada.

O resultado de *speedup* total de execução do algoritmo após o blobo I é mostrado em 4.7.

	N				
Tempo	10^4	10^5	10^6	10^7	10^8
Sequencial	590	1280	7200	69520	699730
Fase 1	600	1340	6390	60130	599690
Speedup	0.98	0.95	1.12	1.15	1.16

Tabela 4.7: *Speedup* total.

4.4 Desempenho com o Bloco I e II

Após a implementação do bloco I, concluímos que algumas funções tiveram seu melhor desempenho na versão sequencial executada pela *cpu*, embora com uma diferença não muito significativa.

4.4.1 Análise dos Trechos Modificados do Bloco II

Número de ocorrências dos trechos do algoritmo que são paralelizáveis é mostrado em 4.11.

	N				
Tipo de Trecho	10^4	10^5	10^6	10^7	10^8
1 (Descrito na subseção 3.7.1)	35	36	34	35	36
2 (Descrito na subseção 3.7.2)	56	55	53	59	59

Tabela 4.8: Número de ocorrências nos tipos de trecho.

O resultado do *speedup* da paralelização do tipo de trecho 1 após o bloco II é mostrado na tabela 4.9.

O resultado do *speedup* da paralelização do tipo de trecho 2 após o bloco II é mostrado na tabela 4.10.

O resultado do tempo total do algoritmo após o bloco II é mostrado na tabela 4.11.

	N				
Tipo de Trecho 1	10^4	10^5	10^6	10^7	10^8
Bloco I	10	30	250	2520	28667
Bloco II	$\simeq$0	20	140	1190	13150
Speedup	$\simeq$0	1.5	1.78	2.11	2.18

Tabela 4.9: *Speedup* do tipo de trecho 1.

	N				
Tipo de Trecho 2	10^4	10^5	10^6	10^7	10^8
Bloco I	$\simeq$0	50	500	5070	51200
Bloco II	10	50	290	3030	29257
Speedup	0	1	1.72	1.67	1.75

Tabela 4.10: *Speedup* do tipo de trecho 2.

	N				
Versão	10^4	10^5	10^6	10^7	10^8
Sequencial	590	1280	7200	69520	699730
Bloco II (Todos os Trechos)	600	1320	6220	57000	570760
Speedup	0.98	0.97	1.15	1.21	1.22

Tabela 4.11: Comparativo geral de resultado *(speedup)*.

O gráfico comparativo de *speedup* é mostrado no gráfico 4.2.

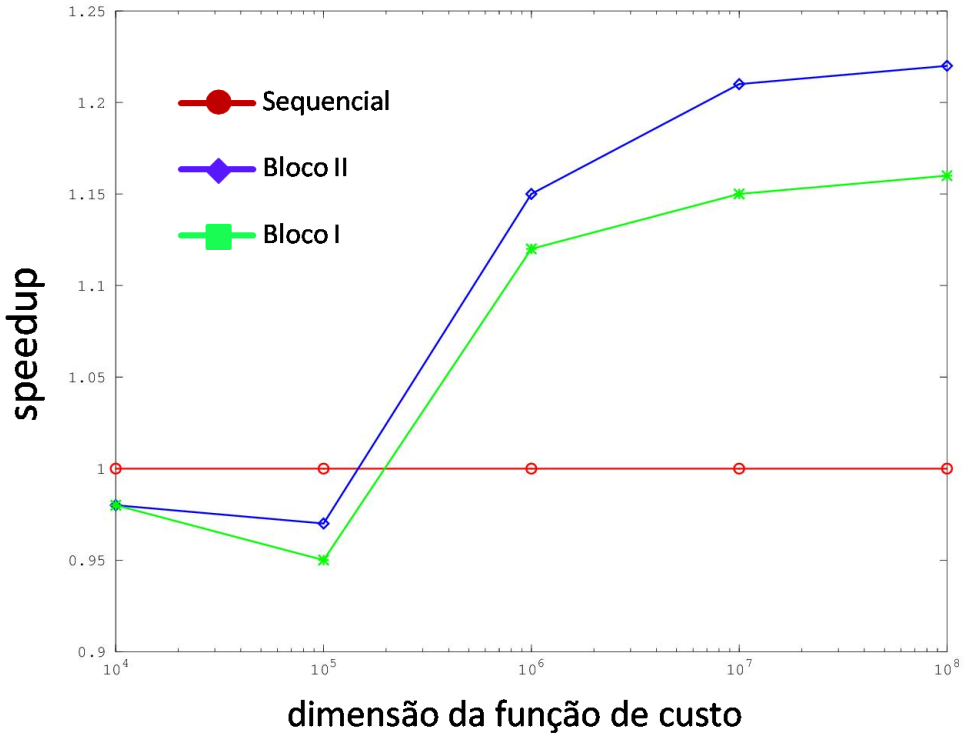

Figura 4.2: Comparativo de *speedup*.

4.5 Comentários Sobre os Resultados

O desempenho da versão modificada do algoritmo L-BFGS apresentou *speedup* maior que 1 (ganho de desempenho) para valores da dimensão da função de custo n maiores que 10^6, e apresentou *speedup* menor que 1 (perda de desempenho) para $n < 10^6$, como constatado na 4.11 e na figura 4.2.

Ressalta-se que a experiência consistiu em paralelizar o L-BFGS sem paralelizar a função de custo, e a opção de uso do cuBLAS nesse caso específico ficou restrito apenas ao nível 1 da mesma.

A implementação do método proposto resultou em aumento significativo de *speedup* nos trechos 1 e 2, como mostrado nas tabelas 4.9 e 4.10, porém no tempo total de execução do algoritmo o ganho de *speedup* não foi maior devido a baixa ocorrência dos trechos como mostrado na tabela 4.8.

As funções *vecadd* e $vecdiff$ tiveram o *speedup* com valores menores que um (perda de desempenho) e por isso sua implementação foi mantida na versão original,

isto é, sequencial, como mostrado na tabela 4.6.

Capítulo 5

Conclusão

Este trabalho analisou e em seguida implementou a versão paralelizada do algoritmo L-BFGS, acelerando-o com o uso de *GPU* e tecnologia CUDA.

O código paralelo foi escrito com o uso da biblioteca cuBLAS. Dessa forma o esforço de desenvolvimento de software foi mitigado, e obteve-se, como esperado, um código final com alto grau de manutenibilidade, pois desenvolve-se software descartando a alta complexidade de escrita em código nativo CUDA.

O exame do código fonte original sequencial do L-BFGS, no qual baseou-se esse trabalho, permitiu identificar uma camada de isolamento com funções vetoriais. Desenvolveu-se a versão paralela das mesmas com aplicação da biblioteca cuBLAS. A implementação desse trecho foi chamado de bloco I.

Prosseguindo na análise do código fonte do L-BFGS verificou-se que era possível atuar fora da camada de isolamento. Propôs-se então um método para adaptação do código nesse local. Essa implementação foi chamada de bloco II.

Os experimentos mostraram que a versão adaptada do código com a técnica do bloco I melhorou o desempenho. A incorporação do método proposto no bloco II conseguiu melhorar em relação ao bloco I, em uma faixa de valores da dimensão da função de custo.

Referências Bibliográficas

[1] AMD. https:/amd.com/pt-br.

[2] Apple. http://www.apple.com/br/.

[3] Xia Cai, Michael R. Lyu, and Kam-Fai Wong. Component-based software engineering: Technologies, development frameworks, and quality assurance schemes.

[4] COMPAQ. http://www.compaq.com/.

[5] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*, volume 1. Morgan Kaufmann, 2012.

[6] CRAY. http://www.cray.com/home.aspx.

[7] KHRONOS GROUP. The open standard for parallel programming of heterogeneous systems.

[8] HP. http://www8.hp.com/br/pt/home.html.

[9] IBM. http://www.ibm.com.

[10] INTEL. http://www.intel.com.

[11] Kamran Karimi, Neil G Dickson, and Firas Hamze. A performance comparison of cuda and opencl. 2010.

[12] Jorge J. More, Burton S. Garbow, and Kenneth E. Hillstrom. Testing unconstrained optimization software. *ACM Transactions on Mathematical Software (TOMS)*, pages 17–41, 1981.

[13] NEC. http://www.nec.com/.

[14] Netlib. https://www.gnu.org/software/gsl/manual/html_node/gslcblas-library.html.

[15] Netlib. http://www.netlib.org/blas/.

[16] J. Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of Computation*, 35 (151)(151):773–782, 1980.

[17] NVIDIA. https://developer.nvidia.com/cublas.

[18] NVIDIA. https://developer.nvidia.com/cublasxt.

[19] NVIDIA. http://www.nvidia.com/object/cuda_home_new.html.

[20] N. Okazaki. liblbfgs: a library of limited-memory broyden-fletcher-goldfarb-shanno (l-bfgs). 2010.

[21] Oracle. http://www.oracle.com/us/sun/index.htm.

[22] Jason Sanders and Edward Kandrot. *CUDA C by Example*, volume 1. Addison Wesley, 2006.

[23] SGI. http://www.sgi.com/.