IntLAG: uma biblioteca para álgebra linear intervalar em processadores gráficos

Ademar Marques Lacerda Filho

Dissertação de Mestrado
em
Ciência da Computação

Universidade de São Paulo
Instituto de Matemática e Estatística
Departamento de Ciência da Computação

Orientador: Prof. Dr. Walter Figueiredo Mascarenhas

O presente trabalho foi realizado com apoio do CNPq, Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil
O autor também recebeu auxílio do projeto 2013/10916-2 da FAPESP na forma de recursos computacionais para realização de experimentos

São Paulo, Maio de 2015

IntLAG: uma biblioteca para álgebra linear intervalar em processadores gráficos

Esta é a versão original da dissertação elaborada pelo
candidato Ademar Marques Lacerda Filho, tal
como submetida à Comissão Julgadora.

Agradecimentos

Devo em primeiro lugar agredecer a minha família por toda a paciência ao longo dos anos e o suporte que deles recebi, em especial meus pais, Ademar e Mercedes. Com certeza sem o incentivo deles à minha educação este trabalho jamais teria sido possível.

Agradeço aos amigos próximos pela convivência e companheirismo nos diversos momentos Ana Ciconelle, Marina Rillo, Victório Neto e Guilherme Augusto. E também aos vários aos amigos de longa data, que apesar da distância física estão sempre presentes: Larissa Souza, Aderbal Neto, Jéssica Medina, Bruno Guimarães, Adônis Penalva e Aída Penalva.

Aos companheiros de laboratório pelas conversas aleatórias, momentos de chá e também dúvidas pertinentes ou não, agradeço ao John, Vítor, Oberlan, Antonio Marcos, Márcia, Anderson e Thilo. Um agradecimento especial dedico ao colega e amigo Tiago Montanher que propôs o tema inicial do projeto e esteve presente durante todo o desenvolvimento do projeto e desta Dissertação.

Agradeço ao Walter Mascarenhas pela orientação que recebi ao longo do trabalho e ao Marco "Gubi" Gubitoso pelas críticas pertinentes em meu exame de qualificação. Um agradecimento especial ao Alfredo Goldman pela disponibilidade em que sempre se apresentou, participação em meu exame de qualificação, pela orientação na etapa final do trabalho, pelo apoio e pelo bom humor.

Resumo

Lacerda Filho, A. M. **IntLAG: uma biblioteca para álgebra linear intervalar em processadores gráficos**. 2014. 120 f. Dissertação de Mestrado - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2014.

A aritmética intervalar é utilizada em problemas físicos e computacionais onde uma solução rigorosa é necessária. Muitos destes problemas são computacionalmente caros e requerem implementações de alto desempenho para que a solução seja viável.

Este trabalho visa a implementação de uma biblioteca de código aberto do tipo BLAS para aritmética intervalar utilizando conceitos de GPGPU por meio da extensão CUDA e a comparação desta com técnicas mais tradicionais.

Apresentamos a biblioteca e sua abordagem focando no aspecto de implementação da mesma. Por fim, apresentamos um *benchmark* comparando o desempenho dos métodos desenvolvidos em GPU com suas contrapartes em CPU, seriais e paralelas, resultados promissores são apresentados em uma placa gráfica de última geração.

Palavras-chave: GPGPU, BLAS, Computação de Alto Desempenho, Aritmética Intervalar, Otimização Numérica.

Abstract

Lacerda Filho, A. M. **IntLAG: Interval Linear Algebra on Graphic Processor Units**. 2010. 120 f. Tese (Doutorado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2010.

Interval arithmetic is used by physical and computational scenarios in which a rigorous solution is required. Many of these problems are computationally expensive and require high-performance implementations.

This work aims to implement an open source BLAS type library for interval arithmetic using GPGPU concepts through CUDA extension and to compare it with more traditional techniques.

We present the library and its approach focusing on the implementation aspects. Finally, we present a benchmark comparing the performance of the methods developed in GPU with their counterparts in CPU, serial and parallel both, promissing results are presented in a high-end GPU.

Keywords: GPGPU, BLAS, High Performance Computing, Interval Arithmetic, Numerical Optimization.

Sumário

Lista de Abreviaturas

ALU	Unidade Lógica e Aritmética (Arithmetic Logic Unit)
API	Application Programming Interface
BLAS	Basic Linear Algebra Subprograms
CPU	Central Processing Unit
GCC	GNU Compiler Collection
GPU	Processador Gráfico (*Graphic Processor Unit*)
GPGPU	Processador Gráfico de Propósito Geral (*General Purpose Graphic Processor Unit*)
IntLAG	Interval Linear Algebra on GPU
IEEE	Institute of Electrical and Electronics Engineers
LAPACK	Linear ALgebra PACKage
MISD	Multiple Instruction Single Data
MIMD	Multiple Instruction Multiple Data
NaN	Not a Number
NVCC	NVidia's CUDA Compiler
OpenMP	Open Multi-Processing
POO	Programação Orientada à Objetos
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SISD	Single Instruction Single Thread
SM	Multiprocessador de Fluxo (Streaming Multiprocessor)
ULP	Unit in Last Place

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Em diversos cenários físicos, matemáticos e computacionais é interessante e necessária a introdução de uma medida de incerteza na representação de escalares. Esta necessidade leva à definição de tipos intervalares e a sua aritmética associada.

A aritmética intervalar *in silico* tem propriedades muito específicas, devido em grande parte aos modos de arredondamento. Além disto, ela é mais cara do que aritméticas usuais que têm implementação em *hardware*. Por esta razão é importante acelerar aplicações que façam uso deste tipo de aritmética.

As arquiteturas de um processadores gráficos (GPUs) fornecem diretivas de hardware interessantes à aritmética intervalar. Além disso, são uma alternativa de paralelismo eficiente e poderosa para aplicações densamente algébricas.

As GPUs modernas possuem muitos núcleos de processamento e se baseiam no modelo SIMD (Single Instruction Multiple Data), onde múltiplos núcleos executam a mesma instrução simultaneamente. Uma das plataformas de programação para GPU é a extensão CUDA, que é admitida somente por processadores gráficos da NVIDIA.

Um BLAS (Basic Linear Algebra Subprograms) é um conjunto específico de rotinas aritméticas altamente paralelizáveis e recorrente a diversas aplicações numéricas, incluindo problemas de aritmética intervalar. Um conjunto de métodos BLAS agrupa um número de funções da algebra linear, como produto escalar e multiplicação de matrizes. Uma biblioteca de rotinas tipo BLAS deve ser altamente otimizada, pois via de regra constituem um gargalo na aplicação.

As rotinas que compõem o BLAS, no geral, são significativamente paralelizáveis em um modelo SIMD. Tal fato e o alto custo de operações aritméticas intervalares sugerem que instâncias de bibliotecas intervalares tipo BLAS em GPU terão um ganho significativo de desempenho para problemas de grandes dimensões.

O desenvolvimento de um pacote deste tipo é o foco deste trabalho. O nome dado a biblioteca implementada foi IntLAG (*Interval Linear Algebra on GPU*).

1.1 Objetivos

O objetivo do trabalho é fornercer uma biblioteca de código aberto tipo BLAS voltada a aritmética intervalar que implemente paralelismo em CPU e em GPU. A implementação tenta equilibrar confiabilidade, usabilidade, legibilidade e desempenho para que seja utilizada como referência em futuros trabalhos.

1.2 Contribuições

A disponibilização da biblioteca em código aberto é a principal contribuição do trabalho. Desconhecemos qualquer outra ferramenta que disponibilize funções tipo BLAS para tipos intervalares em GPU.

O trabalho também compara o desempenho de diferentes implementações para promover um melhor entendimento dos processos subjacentes.

1.3 Trabalhos relacionados

- *Boost Interval Arithmetic Library*[18] é uma biblioteca que implementa tipos intervalares de maneira extensa com diversas funcionalidades: aritmética intervalar básica, comparações de intervalos, funções trigonométricas, modos de arredondamento, etc. Utilizamos as classes intervalares para comparação de desempenho.

- *INTLAB*[25] é provavelmente a bilbioteca de aritmética intervalar mais utilizada historicamente. Esta biblioteca é disponível apenas nos ambientes interativos MATLAB e Octave, e mais recentemente tornou-se paga o que diminui sua atratividade. Além disso, o INTLAB é famoso por sua performance reduzida chegando a frações menores que $1/700$ quando comparado a implementações em $C/C++$, como mostra o próximo item.

- Pál e Scendes[5] compararam o desempenho da implementação de métodos de otimização tipo *branch-and-bound* em MATLAB/INTLAB e algoritmos similares implementados em C. As implementações em C encontradas por eles tiveram *speedups* de 165 a 2106 vezes, sendo a média de 700 vezes mais rápida a implementação em C. Descartamos a comparação com o INTLAB baseada nestes resultados previamente obtidos.

- *IntBLAS*[24] é uma biblioteca de aritmética intervalar serial em C++. A biblioteca é de alto nível, com uma extensa variedade de métodos intervalares e prioriza a usabilidade. Uma adaptação do BLAS é implementada de forma simplificada e não seguindo os padrões de nomenclatura e especificação.

- *GSL CBLAS Library*[27] é uma biblioteca tipo BLAS simples e bastante legível. Boa parte dos métodos BLAS implementados para CPU neste trabalho foram inspirados nas implementações desta biblioteca.

- *MAGMA (Matrix Algebra on GPU and Multicore Architectures)*[26] é um pacote extenso para métodos de algebra linear, incluindo BLAS e LAPACK, para arquiteturas

paralelas, incluindo GPU. Algumas implementações do IntLAG em GPU foram baseadas em implementações do MAGMA.

1.4 Organização do Trabalho

O trabalho está dividido em sete capítulos e dois apêndices. Os Capítulos 2, 3 e 4 é apresentam uma introdução téorica abordando os conceitos básicos e propriedades de paralelismo, arquitetura de processadores gráficos, computação com o modelo CUDA, bibliotecas BLAS, tipos intervalares e sua aritmética particular. O Capítulo 5 aborda os aspectos práticos da implementação da biblioteca IntLAG. No Capítulo 6, discutimos o desempenho da biblioteca apresentando gráficos e tabelas de execuções comparadas, também discutimos o modelo usado para os testes de desempenho. Por fim, o Capítulo 7 explicita as principais conclusões e sugere futuros trabalhos.

Capítulo 2

Computação Paralela em GPU

Durante anos e com grandes investimentos na miniaturização de transistores, foi seguro esperar que a cada período de aproximadamente 18 meses o poder computacional dos processadores dobrassem. No entanto, há bastante tempo, este crescimento tem encontrado uma barreira que tem levado engenheiros a diferentes soluções.

Uma delas é a substituição do silício como principal componente dos transistores visando a continuidade do processo de miniaturização destes componentes. Enquanto a pesquisa avança nesta frente, espera-se que ainda levem muitos anos ou até décadas para que processadores deste tipo cheguem ao mercado.

Outra solução é a adoção em massa de arquiteturas, algoritmos e modelos de paralelismo que se apoiam em múltiplos processadores e a comunicação entre eles para executar uma mesma aplicação. Podemos citar os sistemas *multicore*, coprocessadores Xeon Phi e uma classe de arquiteturas FPGA (Field-Programmable Gate Arrays). O modelo de computação nestas arquiteturas se baseia na criação de múltiplas threads utilizando APIs como *OpenMP*, *Thread Building Block* ou *Posix Threads*.

Outro tipo de plataforma é a arquitetura de processadores gráficos (GPU). As GPUs evoluíram a partir da necessidade de gerar aplicações gráficas de alto nível de tempo real e, a depender da aplicação e do tamanho do problema, podem executar aplicações consideravelmente mais rápidas do que aplicações equivalentes em CPU (Central Processing Unit). Abaixo vemos uma comparação de tempos de execução para multiplicações de matrizes entre diferentes implementações em GPU e uma implementação serial obtida em uma placa da Nvidia[23].

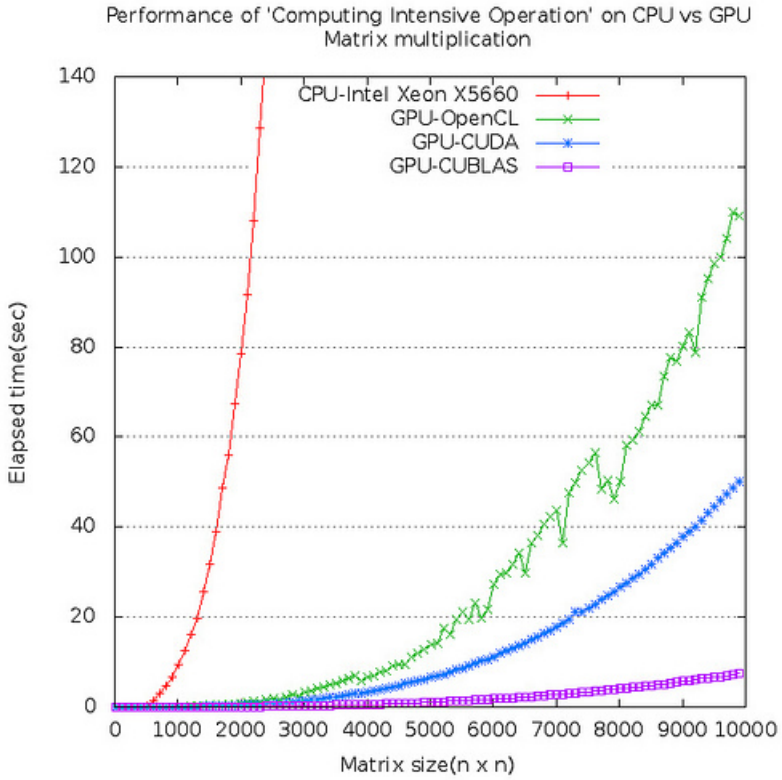

Figura 2.1: *Comparação dos tempos de execução de multiplicação de matrizes quadradas. Tempo de CPU em vermelho contra diferentes implementações de GPU.[23]*

Atualmente, GPUs são sinônimo de desempenho computacional paralelo e vêm ganhando espaço, junto a outros aceleradores no TOP500 e GREEN500[28], projetos que listam os 500 computadores mais poderosos e os mais eficientes em termos energéticos, respectivamente.

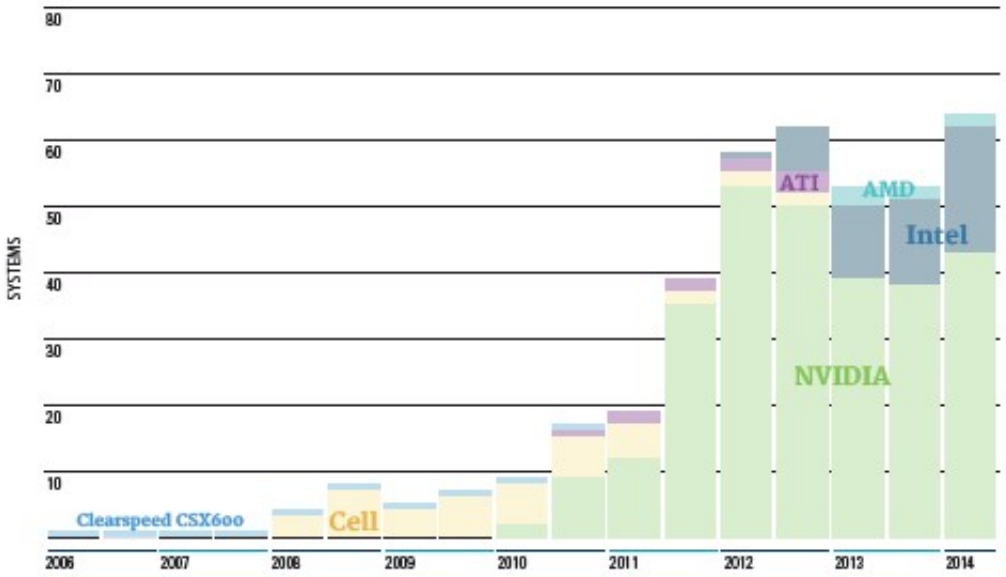

Figura 2.2: *Variação anual do número de computadores do TOP500 que possuem aceleradores e seus fabricantes.[28]*

2.1 Modelos de Paralelismo

2.1.1 Taxonomia de Flynn

Na década de 70, a taxonomia de Flynn foi proposta para classificar arquiteturas de computadores em relação ao seu paralelismo. Essa classificação leva em conta dois fatores: o fluxo de execução de instruções e o fluxo de dados. Apesar de arcaica, a taxonomia de Flynn ainda é usada por sua simplicidade em explicar a relação entre a manipulação de dados e a execução das instruções.

A taxonomia de Flynn reconhece apenas quatro modelos, são eles: SISD, MISD, SIMD, e MIMD.

- SISD (*Single Instruction, Multiple Data*) é o modelo mais simples e que não adota paralelismo. Todas as instruções são executadas por um único processador em um fluxo serial e recebem um único fluxo de dados.
- MISD (*Multiple Instruction, Single Data*) é o modelo menos comum dentre os quatro com poucas aplicações práticas. Neste modelo, múltiplas unidades de processamento operam sobre o mesmo fluxo de dados, resultando em um paralelismo implícito.
- SIMD (*Single Instruction, Multiple Data*) aplica um único conjunto de instruções em múltiplos fluxos de dados. Neste modelo, o paralelismo ocorre quando há independência de dados. Este modelo de paralelismo inspirou as arquiteturas de processadores gráficos.
- MIMD (*Multiple Instruction, Multiple Data*) pressupõe múltiplos fluxos de instruções e dados, sendo portanto o modelo mais genérico. O modelo também requer a independência de dados para seu paralelismo pois a execução é assíncrona. O MIMD é o modelo adotado por arquiteturas *multicore*.

Existem outros modelos não explicitados por Flynn como o SPMD (*Single Program, Multiple Data*) utilizado em sistemas distribuídos e o SIMT (*Single Instruction, Multiple Thread*), modelo utilizado em GPU, derivado do SIMD que será explicado na próxima sessão.

2.1.2 Modelos de Memória

Os modelos de memória em sistemas de computação paralelos descrevem como se dá a interação entre as unidades de processamento e seu uso compartilhado de memória. Modelos de memória definem como o modelo de programação e o compilador devem atuar para manter consistência de dados[19]. Um fator crítico em todos os sistemas de memória é a largura de banda.

Abaixo descreveremos os modelos de memória distribuída e compartilhada. Ressaltamos a existência de modelos mistos. Geralmente, em sistemas mistos, nós de sistemas distribuídos são, por sua vez, sistemas compartilhados.

Memória Compartilhada

Em sistemas de memória compartilhada, cada unidade de processamento têm acesso a um espaço de memória global. Em sistemas deste tipo, há coerência de *cache* e podem haver conflitos de leitura e escrita de dados para diferentes processadores.

O sistema de memória compartilhada mais simples é o UMA (Uniform Memory Access). Nestas arquiteturas, o tempo de acesso a memória é uniforme, pois os processadores estão geometricamente posicionados à mesma distância do módulo de memória.

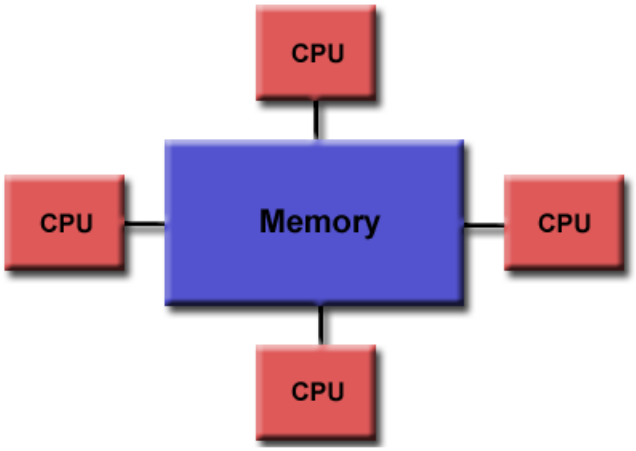

Figura 2.3: *Sistema de memória compartilhada UMA[19].*

Um sistema do tipo NUMA (Non-Uniform Memory Access) é geralmente composto por sistemas UMA ligados por barramentos, como mostra a Figura 2.4. Todos as unidades de processamento têm acesso direto a todos os espaços de memória, no entanto, a geometria da arquitetura costuma não ser uniforme, de modo que certos processadores terão acesso mais rápido a determinados espaços de endereçamento.

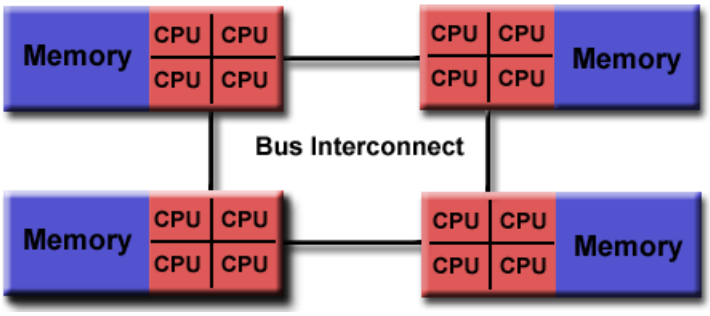

Figura 2.4: *Sistema de memória compartilhada NUMA[19].*

Memória Distribuída

Neste modelo, cada unidade de processamento tem seu espaço de memória local e atua de forma independente. A comunicação em sistemas de memória distribuída é dependente de troca de mensagens em uma rede de comunicação e frequentemente se baseia em pontos de sincronização explícitos.

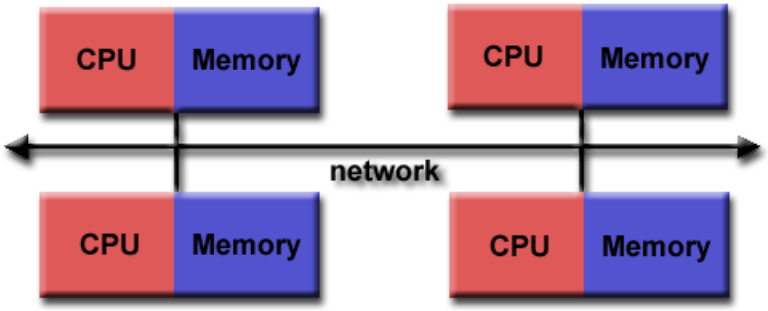

Figura 2.5: *Sistema de memória distribuída[19].*

2.1.3 Speedup

Nas arquiteturas do tipo SISD, podemos modelar o tempo de execução de uma aplicação em função das dimensões dos dados de entrada. Quando consideramos sistemas paralelos, devemos considerar também o número de processadores utilizados e ter em mente os custos de comunicação entre eles.

A principal medida de desempenho em arquiteturas paralelas se chama *speeup*[1] O *speedup* pra um número N de processadores $S(N)$ é definido como a razão do tempo de execução serial $T(1)$ e paralela $T(N)$:

$$S(N) = \frac{T(1)}{T(N)}$$

A *Lei de Amdahl* governa o máximo *speedup* em relação a fração do tempo de execução que é paralelizável e o número de processadores utilizados.

$$S_{max} = \frac{1}{1 - F + \frac{F}{N}}$$

Onde S_{max} é o máximo *speedup* teórico, F é a fração (menor que 1) paralelizável e N o número de processadores da arquitetura.

2.1.4 Taxa de FLOPS

A taxa de FLOPS (Floating Point Operations per Second) é utilizada para comparar o desempenho de diferentes arquiteturas baseada em sua capacidade aritmética de ponto flutuante. Vale notar que FLOPS é uma medida de poder de processamento, sendo a eficiência de algoritmos algo totalmente a parte.

[1] Em português: *aceleração*.

Como arquiteturas atuais têm taxas bastante elevadas, utilizamos múltiplos da unidade FLOPS para nos referirmos a estes sistemas, como GFLOPS (Giga FLOPS[2]) e TFLOPS (Tera FLOPS[3]). Em Novembro de 2014, o supercomputador chinês Tianhe-2 foi listado pelo TOP500 como computador mais rápido do mundo, atingindo a taxa de 33.86 PFLOPS.

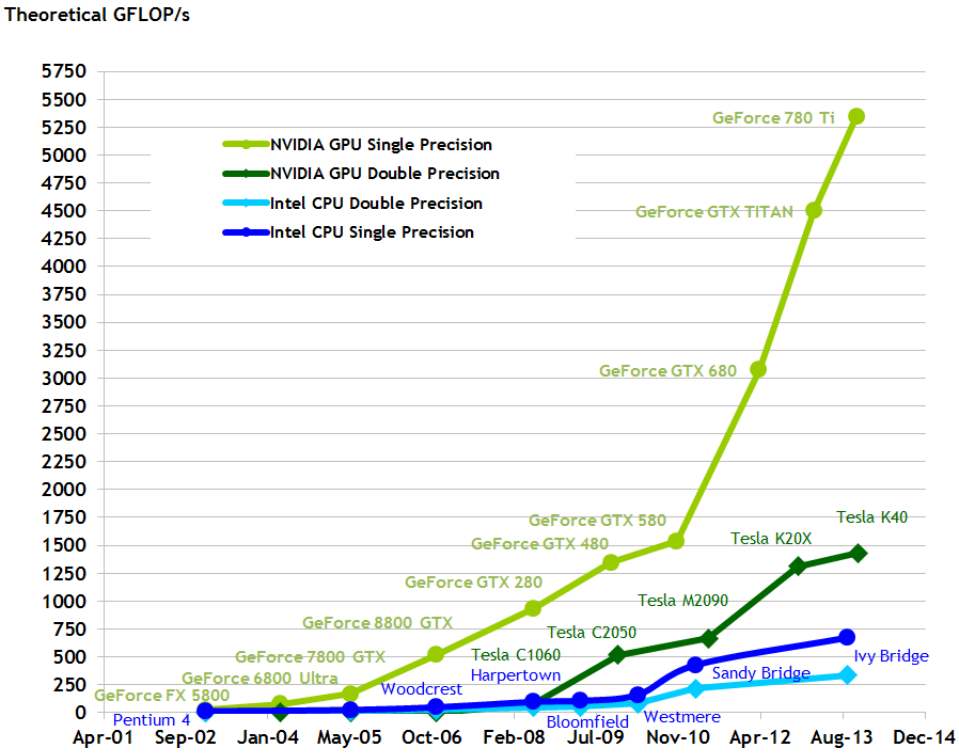

Figura 2.6: *Comparação da taxa teórica de FLOPS em arquiteturas de GPU e CPU para operações de ponto flutuante de precisão simples e dupla[12].*

Arquiteturas de GPU se destacam apresentando um dos maiores índices de FLOPS por valor monetário. Taxas de FLOPS para diferentes arquiteturas são mostradas na Figura 2.6.

2.2 Computação em GPU

2.2.1 Evolução

Antes do advento dos processadores gráficos, os sistemas de computação utilizavam sua principal CPU para gerar toda a interface gráfica, à exceção do sinal ótico. Os primeiros controladores gráficos surgiram na década de 70 em máquinas de fliperama ainda monocromáticas. Ainda nesta década, o Atari 2006 foi o primeiro vídeo game de console a utilizar tais controladores.

Na década de 80, os controladores de vídeo evoluíram agrupando e impulsionando cada vez mais funcionalidades de computação gráfica. Ampliou o espectro de cores, começou a se trabalhar com bitmaps (ao invés de computação bit-a-bit), além do surgimento do primeiro multiprocessador paralelo.

[2]Unidade equivalente a 10^9 FLOPS
[3]Unidade equivalente a 10^12 FLOPS

A década de 90 popularizou os processadores gráficos e continuou trazendo avanços, principalmente em termos de APIs como DirectX e OpenGL. Durante toda a década houve uma grande demanda de gráficos 3D de tempo real, com destaque para a indústria de jogos e algoritmos de renderização *ray tracing* que levam em conta não apenas objetos 3D, mas fatores de luminosidade, sombras, reflexões e transparência.

Figura 2.7: *Imagem gerada por ray tracing[16].*

A primeira GPU como conhecemos hoje foi a Geforce 256 lançada em 1999 pela Nvidia e podia ser programada com OpenGL e DirectX. Desde então, a Nvidia se consolidou como a principal fabricante de GPUs, seguida pela AMD.

Desde a década de 90, já existia interesse da comunidade científica em utilizar processadores gráficos para computação de propósito geral, em particular problemas envolvendo matrizes e vetores naturalmente paralelizáveis como a fatoração LU. Este estilo de computação é designado GPGPU (General-Purpose computing on Graphics Processing Units).

Anteriormente à criação de modelos de programação para GPGPU, esta utilizava artifícios incômodos manipulando estruturas das APIs gráficas disponíveis. Em 2006, a Nvidia lançou a plataforma de programação CUDA (Computer Unified Device Architecture). Em 2009, o OpenCL (Open Computing Language) foi lançado como um alternativa mais genérica.

Juntamente com o CUDA, a Nvidia lançou a GeForce 8800 com 128 núcleos de processamento, a primeira com suporte a GPGPU via CUDA. Baseada nesta arquitetura, a Nvidia lançou a série *Tesla* de GPUs em 2007. A próxima geração foi denominada *Fermi* e visou principalmente o aumento de desempenho e funcionalidades disponíveis. Seguindo a Fermi, em 2012 foi lançada a série Kepler que deu destaque a eficiência energética de suas arquiteturas, em conformidade com a tendência do período.

A Figura 2.8 ilustra a evolução das arquiteturas em relação a funcionalidades e eficiência energética, incluindo a geração atual é a Maxwell e a próxima geração anunciada para 2016, a Pascal.

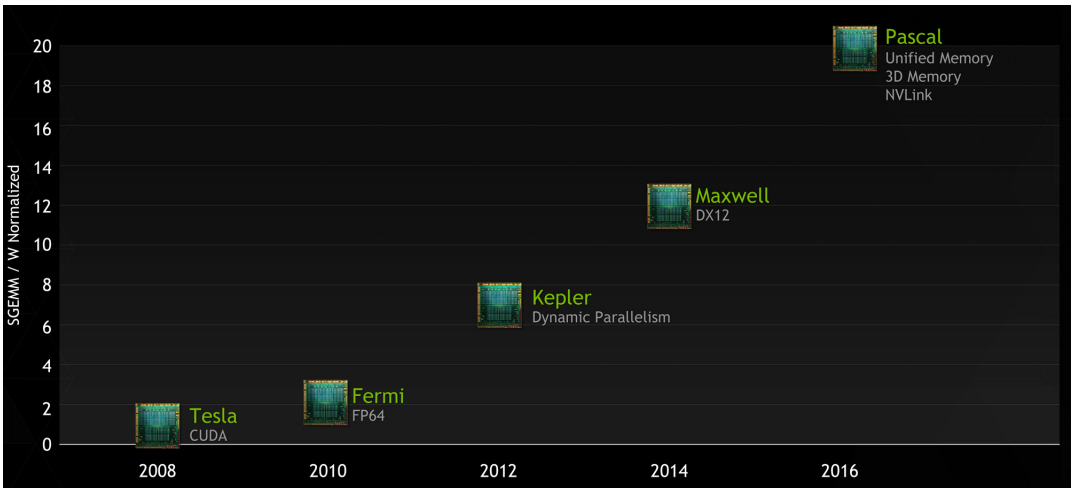

Figura 2.8: *Evolução das arquiteturas Nvidia em relação a eficiência energética[29].*

2.2.2 Arquitetura

As GPUs diferem das CPUs principalmente no número de núcleos básicos de processamento (ALU ou Arithmetic and Logic Unit), como mostra a Figura 2.9. Enquanto ainda é comum encontrar processadores com quatro núcleos, as GPUs possuem de centenas a milhares de ALUs (3072 núcleos na NVIDIA Titan X)[11].

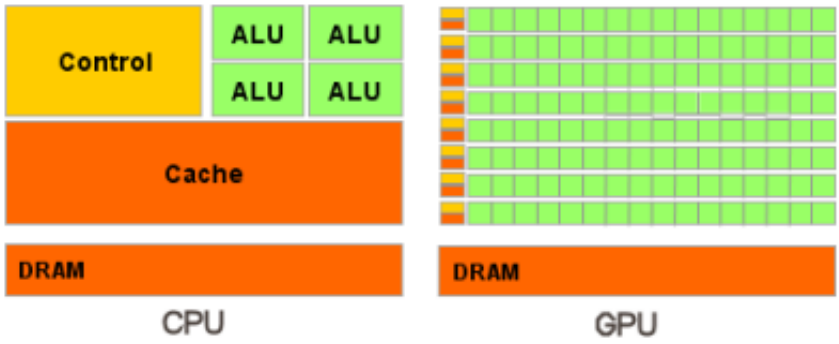

Figura 2.9: *Arquiteturas típicas de uma CPU e uma GPU[12].*

Outra diferença importante é a proporção muito maior de transistores é devotada às ALUs e uma proporção menor às unidades de controle e *cache* em processadores gráficos[12]. Consequências dessa discrepância incluem tempos elevados de leitura e escrita na memória global da GPU, assim como um alto custo de ramificações lógicas, que via de regra devem ser evitadas ao máximo [12]. Porém leva a uma taxa de FLOPS mais elevada.

A Figura 2.10 ilustra a arquitetura Kepler, especificamente os modelos GK110 e GK210. Essa arquitetura possui 15 multiprocessadores (SM)[4]. O multiprocessador é uma unidade de execução contendo diversas ALUs que executarão sempre as mesmas instruções (ou a instrução nula (*nop*) concorrentemente. O multiprocessador possui ainda diversos níveis de

[4]Em inglês: Stream Multiprocessor. A partir da série Kepler a Nvidia passa a usar o termo SMX.

memória local (registradores, memória constante, cache L1), sendo uma unidade de processamento independente.

Figura 2.10: *Arquitetura das placas gráficas Kepler GK110 e GK210[12].*

Cada multiprocessador possui 192 ALUs, em contraste com 32 ALUs na arquitetura Fermi, para um total de 1536 ALUs. Cada ALU pode executar até oito instruções por ciclo, incluindo instruções de precisão dupla que anteriormente eram executadas singularmente.

O código processado serialmente pelas ALUs da GPU são definidos em forma de *kernels*, um tipo especial de função. O programador deve explicitar o número kernels a serem executados por meio das dimenções de um *grid* e dos blocos pertencentes a este *grid*. A Figura 2.11 mostra um *grid* bidimensional (2x3) onde cada bloco pertencente ao *grid* contém 12 *threads* também um formato bidimensional (3x4).

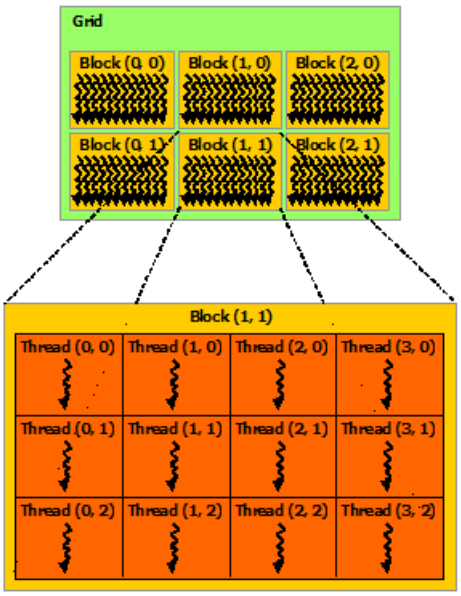

Figura 2.11: *Representação gráfica de um grid bidimensional. Este grid possui três blocos na coordenada x e dois na coordenada y. Além disso, cada bloco possui dimensões 4 x 3 x 1 (x, y e z)[12].*

Um *warp* é a unidade de escalonamento básica. Cada *warp* define um conjunto de 32 *threads* pertencentes a um bloco, mesmo que a dimensão do bloco não seja multipla de 32, *threads* complementares serão criadas e executarão instruções nulas. Cada multiprocessador na arquitetura Kepler possui quatro escalonadores de *warps*, sendo capaz de executar quatro *warps* paralelamente[10].

Cada bloco terá sua execução efetuada em apenas um multiprocessador e portanto terá acesso aos espaços locais de memória durante toda sua execução. A Figura 2.12 ilustra o escalonamento de oito blocos em GPUs com diferentes números de multiprocessadores.

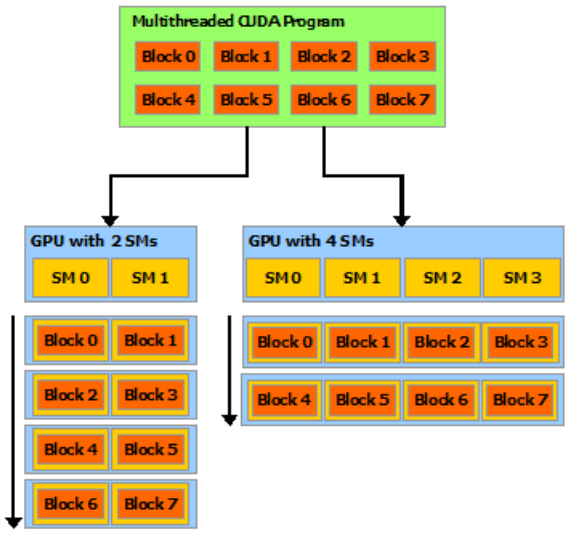

Figura 2.12: *Execução de oito blocos em duas GPUs diferentes. À esquerda, uma GPU com duas SMs e à direita outra com quatro. Execução supõe que os blocos demoram o mesmo tempo para terminar a execução[12].*

A computação em GPGPU, dentro de um multiprocessador, segue o modelo SIMT[10] (Single Instruction Multiple Thread), derivado do SIMD, onde o paralelismo de dados é alcançado por diferentes threads executando as mesmas instruções.

O modelo de memória nas GPUs NVIDIA contempla um espaço de memória global e um cache L2 acessíveis a partir de qualquer *thread* em qualquer *grid*. Há também um pequeno espaço de memória compartilhada acessível por todas as *threads* de um determinado bloco que é compartilhado com o cache L1. Finalmente, cada *thread* terá alocada para si um espaço pequeno de memória local somente acessível dentro da própria *thread* correspondente aos registradores. As latências de acesso à memória em GPU são bastante importantes. Em geral muitas otimizações podem ser feitas usando memória compartilhada cujo acesso é muito menor que um acesso à memória global [11]. Estas relações são ilustradas na Figura 2.13. Outros tipos de memória presentes na GPU são memórias constantes e texturas.

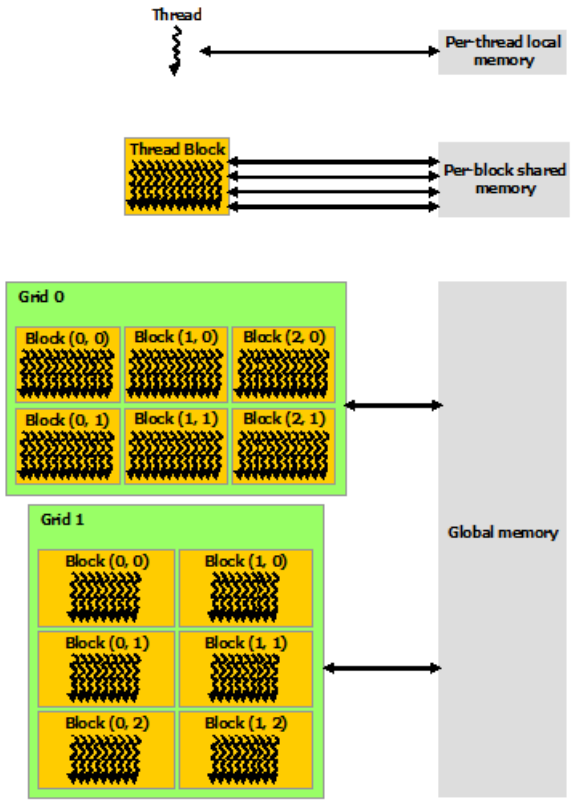

Figura 2.13: *Ilustração de acessos aos vários tipos de memória[12].*

As GPUs mais modernas também possuem dois níveis de cache sendo que o cache L1 é específico para um multiprocessador e o cache L2 é compartilhado por todos eles.

2.2.3 CUDA

CUDA é uma plataforma de programação paralela direcionada a processadores gráficos da Nvidia, podendo ser utilizada apenas por estas arquiteturas. CUDA é uma extensão a diversas linguagens de programação como C, C++, FORTRAN, Java e Python.

As principais vantagens de CUDA sobre OpenCL são a simplicidade, CUDA é de uso muito mais direto e limpo do que OpenCL, e desempenho. Os resultados mais parcimoniosos

[4] indicam que o OpenCL e CUDA podem ter uma performance semelhante, porém indicam também que o *overhead* do OpenCL é muito maior[4].

A principal desvantagem ao uso de CUDA é que é restrito às máquinas com GPUs da Nvidia, excluindo outros fabricantes e placas de vídeo mais antigas. Além do que, em termos financeiros, essas GPUs costumam ser mais caras. Outro ponto importante é que todas as decisões de projeto relacionados a plataforma ficam exclusivamente nas mãos da empresa.

Desde o Compute Capability 2.0 (conjunto de instruções que são suportados pela placa gráfica), CUDA suporta a maior parte do padrão IEEE 754 [14] referente aos modos de arredondamento para aritmética de ponto flutuante de precisão simples e dupla.

A Nvidia utiliza um índice chamado *Compute Capability* para classificar as versões da arquitetura em relação a funcionalidades. Por exemplo, até a versão 1.2 não havia suporte para operações de ponto flutuante de precisão dupla e só a partir da versão 3.5, com a adição de paralelismo dinâmico, existe a possibilidade de um *kernel* disparar outras tarefas para a GPU. As GPUs mais novas estão na versão 5.2 de *Compute Capability*.

Além da versão do hardware, existem as versões da extensão da linguagem CUDA propriamente dita, que define que diretivas podem ou não podem ser chamadas. O CUDA se encontra na versão 7.0. A codificadas de aplicações em CUDA devem ter em mente a versão do CUDA e do *Compute Capability* das arquiteturas em que serão executadas para que se escreva código legalmente executável.

Programação em CUDA

CUDA fornece três prefixos de função que restringem onde as funções podem executar e onde podem ser chamadas. O padrão, quando omitido, é o prefixo __ host__. A tabela 2.1 expõe as relações definidas.

	Execução em	Chamada a partir de
__global __	GPU	GPU / CPU
__device__	GPU	GPU
__ host__	CPU	CPU

Tabela 2.1: *Extensões de funções em CUDA.*

A chamada de um *kernel* é a maneira usual de iniciar a execução de uma função em GPU. Um *kernel* é definido pelo prefixo __global __ e deve ser disparado juntamente com as dimensões de *grid* e bloco desejadas. A listagem abaixo mostra a sintaxe para chamar o *kernel* 'hello' apenas uma vez utilizando CUDA C.

```
1 #include<fenv.h>
2
3 __global__ hello() {
4   printf("Hello!\n");
5 }
6
7 int main() {
8
9 hello<<<1,1>>>();
10
```

```
11 return 0;
12 }
```

Para dimensões maiores, CUDA fornece a função *dim3* que recebe um, dois ou três argumentos numéricos para gerar as dimensões de um *kernel*.

```
1 hello<<<dim3(20,20),dim3(10,15,10)>>>();
```

Uma determinada *thread* pode avaliar seus próprios índices e as dimensão do seu *grid* e bloco para indexar diferentes endereços de memória avaliando estruturas disponíveis no ambiente CUDA:

- *threadIdx* contém as dimensões da *thread* dentro do um bloco.
- *blockIdx* contém as dimensões do bloco dentro de um *grid*.
- *blockDim* contém as dimensões do bloco.
- *gridDim* contém as dimensões do *grid*

Cada uma destas quatro estruturas podem ser avaliadas em suas dimensões x, y e z. Por exemplo: *threadIdx.y*.

A alocação de memória em GPU e a transferência dos dados entre a memória da CPU e da GPU também devem ser explicitas. Essas operações são realizadas com as funções fornecidas por CUDA *cudaMalloc, cudaMemcpy e cudaFree*. Listaremos a seguir o exemplo clássico da soma de vetores em CUDA para 900 elementos que gerencia memória e utiliza indexação no *kernel*.

```
1 __global__ vadd(int* a, int* b, int* c) {
2
3   short i = blockIdx.x *blockDim.x + threadIdx.x;
4
5   a[i] = b[i] + c[i];
6 }
7
8 //Operacao vetorial a = b + c em GPU
9 int main() {
10
11   int *a, *b, *c;
12   int *d_a, *d_b, *d_c;
13
14   // Aloca e preenche os valores a, b, c
15   ...
16
17   cudaMalloc((int **)&d_a, 900);
18   cudaMalloc((int **)&d_b, 900);
19   cudaMalloc((int **)&d_c, 900);
20
21   cudaMemcpy(d_b, b, 900, cudaMemcpyHostToDevice);
22   cudaMemcpy(d_c, c, 900, cudaMemcpyHostToDevice);
```

```
23    ...
24    vadd<<<30,30>>>(d_a, d_b, d_c);
25
26    cudaMemcpy(a, d_a, 900, cudaMemcpyDeviceToHost);
27
28    cudaFree(d_a);
29    cudaFree(d_b);
30    cudaFree(d_c);
31
32    // Free a, b, c
33
34    return 0;
35 }
```

Capítulo 3

Aritmética Intervalar

A aritmética intervalar surgiu entre o fim da década de 50 e o começo da década de 60. Os primeiros trabalhos neste campo foram de Sunaga e Moore[9].

A motivação inicial para a criação da aritmética intervalar foi a estimação de erros cometidos em longas sequências de cálculos computacionais devido a erros de arredondamento. Esta necessidade surgiu concomitantemente com o advento dos computadores e necessidade de estimar erros em cálculos científicos.

Muitas outras aplicações surgiram que utilizam a aritmética intervalar. Entre eles estão aplicações gráficas, como cálculo de intersecção de retas, provas matemáticas auxiliadas por computadores, onde podemos definir regiões espaciais e calcular propriedades em regiões de interesse, métodos de otimização, onde intervalos contendo máximos de funções podem ser encontrados por métodos numéricos e computação verificada.

3.1 Motivação e Definição de Tipos Intervalares

Consideremos a lei de Ohm escrita na forma $V = R * I$, onde V representa a diferença de potencial em um condutor, R é a resistência deste condutor e I é a corrente elétrica medida. Se sabemos que a resistência é 1.0 ± 0.2 kilo-ohms e medirmos I como 10 ± 0.1 mili-amperes podemos afirmar que a diferença de potencial é 10.02 ± 2.1 Volts. Alternativamente, podemos dizer que o valor da diferença de potencial em Volts pertence ao intervalo [7.92, 12.12].

A figura 1 abaixo, mostra como uma função real pode ser representada em um domínio discreto por meio de seus ínfimos e supremos em conjuntos discretos.

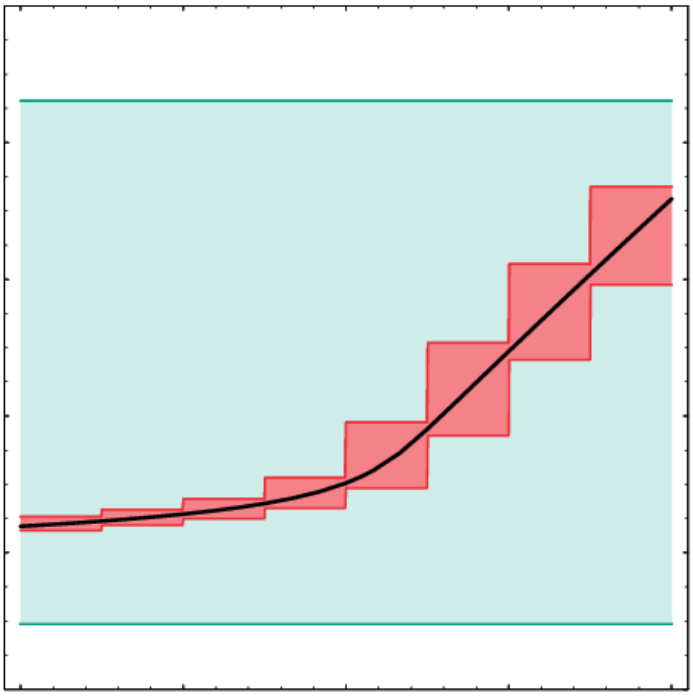

Figura 3.1: *Representação de uma função real em uma partição do domínio por meio de intervalos representados na ordenada[30].*

Os exemplos acima utilizam intervalos para representar escalares sujeitos a uma medida de incerteza. No exemplo da Lei de Ohm, a incerteza deriva da impossibilidade de mensurar qualquer grandeza do mundo físico com precisão absoluta. Outra fonte de imprecisão, agora computacional, é a impossibilidade da fiel representação de um número real genérico *in silico* pela limitação do número de *bits* que uma dada arquitetura admite.

3.2 Operações Básicas

Intervalos são comumente representados por seus limitantes superior e inferior na forma de pares ordenados limitados por colchetes[9]:

$$x = [\underline{x}, \overline{x}],$$

onde o intervalo x denota o conjunto $\{x \in \mathbb{R} : \underline{x} \leq x \leq \overline{x}\}$. Reiteramos que os intervalos como definimos são fechados, ou seja, os extremos fazem parte do conjunto. Uma representação alternativa, como no exemplo do condutor, é a representação centro-raio [9].

Chamamos atenção ao caso específico em que um intervalo representa um número real: o intervalo degenerado. Chamamos de intervalo degenerado qualquer intervalo onde $\underline{x} = \overline{x}$, que contém um único número real, podendo ser representado em ambas as notações e mantendo suas propriedades intervalares.

A partir dos extremos de um intervalo, podemos calcular várias características secundárias do mesmo, alguns exemplos são: centro, raio, comprimento e norma. Operações binárias

de conjuntos também podem ser trivialmente definidas, como: intersecção, união e *hull*[1] Para a definição destas operações consulte o capítulo 2 de Moore et al..

Dois intervalos x e y são iguais se e somente se seus extremos são iguais:

$$x = y \iff \underline{x} = \underline{y} \text{ e } \overline{x} = \overline{y}$$

No entanto, as relações de ordem ($<$ e $>$) não são bem definidas quando a intersecção dos operandos é não nula.

Uma das bases da aritmética intervalar é a propriedade inclusiva respeitada pela maioria das funções intervalares. Informalmente podemos dizer que a propriedade inclusiva garante que a aplicação de funções ou operações aplicadas a intervalos resultem no menor intervalo contendo todos os reais resultantes da mesma operação todos as combinações de reais contidos nos intervalos correspondentes.

Definimos abaixo as quatro operações fundamentais, que respeitam a propriedade inclusiva, na forma:

$$x \oplus y := \{a \oplus b : a \in x, biny, a \in \mathbb{R}, b \in \mathbb{R}\}$$

- Soma

$$x + y = [\underline{x}, \overline{x}] + [\underline{y}, \overline{y}] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$$

- Subtração

$$x - y = [\underline{x}, \overline{x}] - [\underline{y}, \overline{y}] = [\underline{x} - \underline{y}, \overline{x} - \overline{y}]$$

- Multiplicação

$$x * y = [\underline{x}, \overline{x}] * [\underline{y}, \overline{y} = [min(\underline{x} * \underline{y}, \underline{x} * \overline{y}, \overline{x} * \underline{y}, \overline{x} * \overline{y}), max(\underline{x} * \underline{y}, \underline{x} * \overline{y}, \overline{x} * \underline{y}, \overline{x} * \overline{y})]$$

- Inversa e Divisão

  Definimos a divisão a partir da inversa de um intervalo, onde o intervalo invertido não contém 0[2].

$$\frac{1}{y} = [\frac{1}{\underline{y}}, \frac{1}{\overline{y}}], 0 \notin y$$

$$\frac{x}{y} = x * \frac{1}{y}, 0 \notin y$$

[1] *Hull*, ou envoltório é o menor intervalo que contém ambos os intervalos operandos.

[2] Existem maneiras de se definir uma divisão estendida que permite a divisão por intervalos que contêm o 0, porém elas admitem intervalos abertos cujos limitantes são infinitos e união de intervalos disjuntos[9]

A partir das definições de soma e produto, é possível demonstrar propriedades de intervalos definidos sobre $\mathbb{R}$.

Sejam x e y e z quaisquer intervalos definidos sobre $mathbbR$, e sejam os intervalos degenerados **0** e **1** os únicos que satisfaçam as propriedades abaixo, pode-se derivar as nove seguintes propriedades algébricas:

$$x + y = y + x \text{ (Comutatividade da soma)}$$

$$x * y = y * x \text{ (Comutatividade da multiplicação)}$$

$$(x + y) + z = x + (y + z) \text{ (Associatividade da soma)}$$

$$(x * y) * z = x * (y * z) \text{ (Associatividade da multiplicação)}$$

$$x + y = x + z \implies y = z \text{ (Cancelamento da Soma)}$$

$$x + 0 = x \text{ (Identidade da Soma)}$$

$$x * 1 = x \text{ (Identidade da Multiplicação)}$$

$$x * 0 = 0 \text{ (Existência de elemento nulo)}$$

$$x * (y + z) \subseteq x * y + x * z \text{ (Subdistributividade da soma pelo produto)}$$

Ao mesmo tempo, não valem as propriedades de elementos inversos, ou seja, nem sempre existem x ou y satisfazendo:

$$x + (-x) = 0$$

$$x * y = 1$$

Portanto, vemos que a aritmética intervalar não satisfaz as propriedades da aritmética real e que o espaço de intervalos sobre reais não constitui um espaço linear. Ainda assim, muitas boas propriedades podem ser derivadas para que a aritmética intervalar seja útil na prática. Por exemplo, condições suficientes para existência e unicidade de solução de sistemas lineares podem ser definidas[9].

3.3 Modos de Arredondamento

Do ponto de vista computacional, há um ponto sensível de grande interesse: o arredondamento de ponto flutuante. Apenas um subconjunto da reta real é representável em determinada arquitetura, portanto, ao se calcular um número real ele deve ser mapeado por meio de uma aproximação para alguma representação neste conjunto[11].

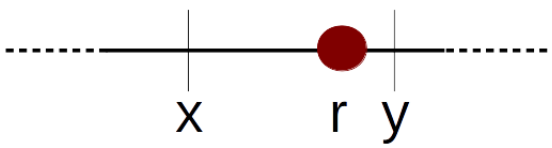

Figura 3.2: *Real r, deve ser arredondado para x ou y[8].*

O padrão IEEE754[14] para operações de ponto flutuante, define quatro operações de arredondamento: em direção ao zero, em direção a ambos valores infinitos e arredondamento pelo mais próximo. Quando o operando não é representável na arquitetura, o arredondamento deve escolher entre os valores imediatamente inferior ou superior, que são separados por uma medida chamada ulp (unit in the last place)[11].

No arredondamento de intervalos, o cálculo do limitante inferior deve sempre ser arredondado para baixo e no cálculo do limitante superior o contrário é verdadeiro, ao que se chama de *outward rounding* ou arredondamento “para fora". Tal arredondamento é necessário para garantir que o escalar representado está contido no intervalo resultante. Infelizmente o padrão não define este tipo de aproximação, e ele deve ser obtido a partir dos quatro tipos existentes.

Nas linguagem $C/C++$, o controle do modo de arredondamento é possível por meio da biblioteca padrão *fenv.h*. A maneira correta de se fazer este controle para uma operação específica é guardar o estado de arredondamento anterior, trocar pelo desejado e restituir o estado anterior após os cálculos desejados. Abaixo um exemplo em $C++$:

```
1 #include<fenv.h>
2
3 int main() {
4
5 // codigo utilizando arredondamento padrao para o mais proximo
6 ...
7
8 int r = fegetround();
9 fesetround(FE_DOWNWARD);
10
11 // codigo utilizando arredondamento para baixo
12 ...
13
14 fesetround(r);
15
16 // codigo utilizando arredondamento padrao para o mais proximo
17 ...
18
19 }
```

A operação de troca de modo de arredondamento mencionada acima é bastante custosa, pois pressupõe uma troca de contexto da CPU. Montanhe comparou mudanças do modo de arredondamento com multiplicações de ponto flutuante. Os resultados obtidos mostraram que a mudança do modo de arredondamento é de 10 a 14 vezes mais custosa que a multiplicação[8].

CUDA também implementa o padrão de arredondamento como especificado pelo padrão IEEE754. Porém, ao invés de implementar troca de contextos nos multiprocessadores, a Nvidia fornece primitivas de *hardware* que melhoram em muito o tempo de processamento. Abaixo um exemplo de código:

```
1
2 __global__ void round_operations(float x, float y) {
3
4   x = __fadd_rd(x, y); // operacao x+y arredondada para baixo
5
6   y = __fmul_ru(x, y); // operacao x*y arredondada para cima
7
8   x = __fsqrt_rz(x); // raiz quadrada de x truncada (arredondada em
        direcao ao zero)
9
10  x = __fdiv_rn(x, 2.0f); // operacao x/y arredondada para o mais proximo
11 }
```

Definiremos agora as quatro operações fundamentais para intervalos levando em conta o modo de arredondamento. Operação arredondada para cima serão denotadas dentro do operador $\uparrow()$, e operações arredondadas para baixo dentro do operador $\downarrow()$:

- Soma

$$x + y = [\downarrow(\underline{x} + \underline{y}), \uparrow(\overline{x} + \overline{y})]$$

- Subtração

$$x - y = [\downarrow(\underline{x} - \underline{y}), \uparrow(\overline{x} - \overline{y})]$$

- Multiplicação

$$x * y = [\downarrow(min(\underline{x} * \underline{y}, \underline{x} * \overline{y}, \overline{x} * \underline{y}, \overline{x} * \overline{y})), \uparrow(max(\underline{x} * \underline{y}, \underline{x} * \overline{y}, \overline{x} * \underline{y}, \overline{x} * \overline{y}))]$$

- Divisão

$$\frac{x}{y} = [\downarrow(min(\frac{\underline{x}}{\underline{y}}, \frac{\underline{x}}{\overline{y}}, \frac{\overline{x}}{\underline{y}}, \frac{\overline{x}}{\overline{y}})), \uparrow(max(\frac{\underline{x}}{\underline{y}}, \frac{\underline{x}}{\overline{y}}, \frac{\overline{x}}{\underline{y}}, \frac{\overline{x}}{\overline{y}})], 0 \notin y$$

Assim, garantimos a consistência das operações intervalares em ambiente computacional, apesar de obtermos intervalos mais largos do que suas contrapartes teóricas.

3.4 Padrão IEEE

No início de 2008, foi criado o grupo de trabalho P1788 da IEEE[7] para definir um padrão para aritmética intervalar computacional, o grupo ainda está ativo. O padrão se baseia no Padrão IEEE 754 de aritmética de ponto flutuante.

Os principais objetivo do padrão são o aumento de portabilidade das bibliotecas e a confiabilidade dos resultados. As propostas incluem a definição dos tipos intervalares básicos, o uso mandatório de modos de arredondamento, as definições das operações fundamentais, o tratamento de intervalos infinitos e limitações nos erros admitidos em certos cálculos[6].

Capítulo 4

Métodos Básicos da Álgebra Linear

Um BLAS (Basic Linear Algebra Subprograms) é um conjunto de funções de álgebra linear de baixo nível, tipicamente utilizados por vários métodos de alto nível. Um subconjunto do que hoje chamamos de BLAS, foi proposto primeiramente por Lawson et al. em 1979 [2] juntamente com uma implementação em FORTRAN.

Estas primeiras funções nada mais eram do que simples implementações de métodos básicos de álgebra linear envolvendo apenas um ciclo de iterações, como produto escalar e soma de vetores. Com o passar dos anos, métodos de ordem mais alta envolvendo dois ou três tais ciclos encaixados foram adicionados ao que hoje chamamos de BLAS [3], por exemplo, multiplicações de matriz-vetor e matriz-matriz e resoluções de sistemas lineares triangulares.

Pacotes BLAS em geral são responsáveis por grande parte da performance de ambientes de computação numérica, como MATLAB, Mathematica, Numpy e R. Uma vez que um pacote BLAS é crítico nestes ambientes e em programas que manipulem entidades vetoriais, é de grande importância desenvolver pacotes que sejam altamente otimizados. Grandes empresas de hardware, como Intel e AMD disponibilizam tais pacotes, procurando suprir esta necessidade.

4.1 Métodos

O nível um do BLAS especifica as operações entre vetores. Os métodos deste nível são: cópia, troca (*swap*), multiplicação de vetor por escalar, soma de vetores, soma dos elementos de um vetor, produto escalar, norma dois, rotações de Givens e operações de máximos e mínimos.

No nível dois, existem apenas três tipos de operações: multiplicação matriz-vetor, solução de sistema linear triangular e operações do tipo *rank-update*. Estas operações estão definidas para matrizes gerais, hermitianas (nos casos de operações com complexos), triangulares e simétricas, sendo as matrizes dadas na forma usual, compactada ou por bandas (exemplificadas em seguida), onde os elementos subentendidos das matrizes triangulares e simétricas são ignorados.

O BLAS nível 3 especifica as mesmas operações do BLAS nível dois, porém para matriz-matriz e sem uso formas compactadas.

Considere a matriz simétrica, onde os elementos da diagonal inferior são ignorados, abaixo:

$$\begin{pmatrix} a & b & c \\ b & d & e \\ c & e & f \end{pmatrix}$$

A forma compactada da matriz é o vetor

$$\begin{pmatrix} a & b & c & d & e & f \end{pmatrix}$$

). Onde, também nas estrutura correspondentes a matrizes triangulares e hermitianas, a porção inferior das matrizes também é ignorada.

Na estrutura compacta que representa matrizes de bandas, cada diagonal a ser considerada é dada por uma coluna da matriz compactada:

$$\begin{pmatrix} 0 & a & b \\ c & d & e \\ f & g & 0 \end{pmatrix}$$

Esta forma corresponde a matriz de bandas abaixo, com uma banda superior e uma banda inferior:

$$\begin{pmatrix} a & b & 0 \\ c & d & e \\ 0 & f & g \end{pmatrix}$$

4.2 Especificações Gerais

Pacotes BLAS como Intel MKL, OpenBLAS e cuBLAS[21] adotam uma nomenclatura específica em suas funções [3]. Por exemplo, a multiplicação genérica de matrizes tem o sufixo *gemm* (GEneral Matrix Multiplication), a multiplicação matriz por vetor tem o sufixo *gemv* (GEneral Matrix Vector), o produto escalar tem o sufixo *dot* (DOT product).

Os prefixos de cada método costumam denotar o tipo de variáveis utilizadas pelas estruturas (inteiros, precisão simples, precisão dupla, etc...), que pode ser eliminado com o uso de programação genérica. Todas estas convenções de nomenclatura facilitam a utilização por usuários de múltiplos pacotes do tipo BLAS.

Além da nomenclatura[22], os padrões definem certos aspectos das operações que devem ser passados como parâmetros. Alguns destes aspectos são listados abaixo:

- Incrementos - os incrementos representam a unidade de espaçamento nos vetores e matrizes que deve ser incrementada em cada operação. Eles descrevem o subconjunto dos *arrays* que devem ser utilizados no cálculo, preservando os demais.
- *trans* - denota se as matrizes devem ser transpostas para a operação. Este parâmetro permite otimização dos algoritmos, pois a transposição de fato não é necessária, apenas uma troca de indíces.

- *side* - este parâmetro é utilizado em algumas operações matriz-matriz para indicar a permutação de certas multiplicações ($A * BouB * A$), onde a primeira matriz conserva outras propriedades e, portanto, uma simples permutação de parâmetros não especifica a totalidade da operação.

- *uplo* - este parâmtro indica qual parte das matrizes triangulares e simétricas devem ser utilizadas para acesso e modificação. O restante da matriz não deve ser acessada.

- *diag* - é um parâmetro usado apenas nos métodos que tratam matrizes triangulares. Ele indica se a matriz é triangular unitária (todos os elementos da diagonal principal tem valor unitário), sendo estes elementos subentendidos.

Capítulo 5

A Biblioteca IntLAG

O IntLAG (*Interval Linear Algebra on GPU*) é uma biblioteca para aritmética intervalar que implementa funções do tipo BLAS e utiliza recursos computacionais de GPU. A biblioteca foi desenvolvida utilizando C++ com as extensões CUDA e OpenMP(Open Multiprocessing).

A biblioteca fornece operações básicas da aritmética intervalar a partir da definição de classes de intervalos, representados pelos seus ínfimos e supremos, classes que abstraem *arrays* de intervalos (chamadas "caixas") e a implementação de um conjunto de métodos BLAS. A biblioteca traz implementações alternativas à CPU e à GPU. Todo o código do pacote é encapsulado em um *namespace* chamado "intlag".

O pacote faz uso de técnicas de Programação Orientada a Objetos(POO), Programação Paralela e Programação Genérica. POO é utilizada na definição das classes e interfaces por toda a biblioteca. Conceitos de Programação Paralela são utilizados largamente nos métodos BLAS que utilizam OpenMP e CUDA.

Programação Genérica é utilizada por meio de *templates* para abstrair funcionalidades comuns de diferentes estruturas que devem ser instanciadas em tempo de compilação das aplicações. Os tipos básicos especializados na biblioteca para intervalos são apenas *float*[1] e *double*[2]. O IntLAG faz uso extenso de programação genérica, o que resulta em uma redução drástica de duplicação de código.

A biblioteca está disponível no GitHub no endereço: https://github.com/ademarl/intlag. As funcionalidades da biblioteca estão na pasta ***include*** e para utilizá-la, basta adicionar o arquivo ***include/intlag.h***.

Dependências

O IntLAG foi desenvolvido exclusivamente para sistemas Linux. A biblioteca depende de compiladores GCC, versões 4.6 ou superior. A biblioteca MPFR(*Multiple Precision Floating Point*)[3] é utilizada para cálculo da função raíz quadrada em CPU, por sua vez, esta depende

[1] Ponto flutuante de precisão simples
[2] Ponto flutuante de precisão simples
[3] *libmpfr-dev*

dos Pacotes GMP[4] e MPC[5] que devem ser propriamente instalados para utilização do código serial.

Para o código dependente de CUDA, recomenda-se, o compilador NVCC mais recente da Nvidia. As placas de vídeo da série Tesla, com Compute Capability menor que 2.0 não são suportadas pelo IntLAG, pois não implemetam operações de ponto flutuante de precisão dupla e certas diretivas de *hardware* utilizadas, como o FMA(*Fused Multiply Add*).

O *GoogleTest 1.7.0* é utilizado para testes de unidade e como base para os testes de desempenho. A biblioteca intervalar do Boost é utilizada para comparação no *benchmark*. Ambos os pacotes são disponibilizados junto ao IntLAG.

5.1 Intervalos

Os intervalos constituem o tipo aritmético básico utilizado pelo BLAS da IntLAG. Dois tipos de intervalos são utilizados, o Interval com implementação em CPU e o CudaInterval, implementado em GPU.

A classe Interval e seus métodos está definida em ***include/interval/interval.h*** e ***include/interval/interval_lib.h***. Os arquivos que definem a CudaInterval e suas operações são ***include/cuda_interval/cuda_interval.h*** e ***include/cuda_interval/cuda_interval_lib.h***

A estrutura de ambas as classes é bastante similar, constituindo a instância de maior duplicação de código na biblioteca. O que motivou esta decisão de projeto foi uma preferência pela não utilização de herança no começo do desenvolvimento.

Os intervalos são definidos computacionalmente pelos seus extremos:

```
1 template <class T>
2 class Interval {
3   public:
4     ...
5
6
7     T const &inf() const;
8     T const &sup() const;
9
10  private:
11    T inf_;
12    T sup_;
13 };
```

As classes intervalares definem ainda diversos construtores, comparações de igualdade e desigualdade, operações de atribuições e um método boleano para verificar se o intervalo é vazio. Um intervalo vazio, é definido na biblioteca como tendo um dos extremos igual ao tipo numérico NaN(Not a Number).

A biblioteca não implementa intervalos abertos com extremos infinitos. Outras operações que não são suportadas são as comparações entre intervalos em relação à ordem pois a propriedade de tricotomia não é garantida quando a intersecção de intervalos não é vazia[6].

[4] *libgmp-dev*
[5] *libmpc3*
[6] É possível definir as operações de ordem para intervalos de maneira ternária, onde um operando pode

Métodos diversos de interesse são definidos para tipos intervalares. Entre eles estão as operações boleanas *contain_zero* e *empty*, que respectivamente respondem se o intervalo contém o zero e se é vazio[7], funções que retornam o comprimento (*width*) e a mediana(*median*) dos intervalos.

Outros métodos binários que retornam um intervalo correspondete são o *overlap* (intersecção), hull[8] (envoltório ou casca), raíz quadrada (para intervalos estritamente positivos) e *abs* que retorna o valor absoluto do intervalo. A definição do valor absoluto é dada abaixo:

$$abs(x) := [0, max(-\underline{x}, \overline{x})], \text{ se } 0 \in x$$
$$abs(x) := [min(-\overline{x}, \underline{x}), max(-\underline{x}, \overline{x})], \text{ se } 0 \notin x$$

A IntLAG implementa os operadores binários + e − segundo as definições:

$$+x := x$$
$$-x := [-\overline{x}, -\underline{x}]$$

As operações de soma, subtração, produto e divisão[9] são implementadas por meio da sobrecarga de operadores incluindo operações com tipos escalares básicos além de apenas operações intervalo-intervalo. Abaixo vemos o código fonte da multiplicação de intervalos:

```
1 template <class T> inline
2 Interval<T> operator*(Interval<T> const x, Interval<T> const y)
3 {
4   return Interval<T>(  min( Rounder<T>::mul_down(x.inf(), y.inf()),
5                             Rounder<T>::mul_down(x.inf(), y.sup()),
6                             Rounder<T>::mul_down(x.sup(), y.inf()),
7                             Rounder<T>::mul_down(x.sup(), y.sup()) ),
8                        max( Rounder<T>::mul_up(x.inf(), y.inf()),
9                             Rounder<T>::mul_up(x.inf(), y.sup()),
10                            Rounder<T>::mul_up(x.sup(), y.inf()),
11                            Rounder<T>::mul_up(x.sup(), y.sup())) );
12 }
```

As operações de máximos e mínimos utilizadas nas implementações destes métodos se encontram no arquivo ***include/aux/min_max.h*** que utilizam diretivas de hardware ao invés de condições *if-else*.

Para a consistência das operações intervalares são definidas duas classes que funcionam como módulos de funções para operações com modos de arredondamentos: Rounder e CudaRounder.

A classe Rounder utiliza o *Floating-Point Environment* (fenv) da biblioteca padrão para mudar os modos de arredondamento em CPU para as operações básicas. Uma estrutura e duas funções acessoras auxiliares são incluídas do arquivo ***include/interval/round_status.h*** para facilitar o uso do ambiente, o código deste arquivo está disponível no Apêndice A. Um segmento de código da classe é mostrado abaixo como ilustração:

ser maior, menor ou intersectar o outro[9].

[7] É interessante definir o método global além do método de classe

[8] *Hull* é definido como a intersecção de todos os intervalos que contém ambos os operandos.

[9] A divisão é definida apenas para intervalos que não contém o zero. Caso contenha o zero, um intervalo vazio é devolvido.

```
1 template<class F>
2 class Rounder {
3   public:
4   ...
5
6   inline static F mul_up(const F x, const F y)
7   {
8     short oldstatus = getRoundMode();
9     setRoundMode(RoundStatus::up);
10    F result = x * y;
11    setRoundMode(oldstatus);
12    return result;
13  }
14
15  inline static F mul_down(const F x, const F y)
16  {
17    short oldstatus = getRoundMode();
18    setRoundMode(RoundStatus::down);
19    F result = x * y;
20    setRoundMode(oldstatus);
21    return result;
22  }
23
24  ...
25 };
```

A classe CudaRounder, como dito anteriormente, utiliza primitivas de hardware para atingir o mesmo propósito da classe Rounder. Nela, no entanto, é necessário especializar as classes para *float* e *double*, pois estas aritméticas de ponto flutuante têm diferentes diretivas. Abaixo o código equivalente as operações de multiplicação para a especialização da classe em precisão simples:

```
1 template<>
2 class CudaRounder<float> {
3   public:
4   ...
5
6     __device__ inline static
7     float mul_down(const float x, const float y)
8     {
9         return __fmul_rd(x, y);
10    }
11
12    __device__ inline static
13    float mul_up(const float x, const float y)
14    {
15        return __fmul_ru(x, y);
16    }
17
18  ...
19 };
```

5.2 Alto Nível

Um dos objetivos originais do IntLAG foi a fácil usabilidade por usuários acostumados ao ambiente MATLAB. A escolha de linguagem C++, dentre outros motivos, se deu pela liberdade de definição de operadores matemáticos utilizando Programação Orientada a Objetos, permitindo que o usuário utilize operadores de forma intuitiva ao invés de um estilo funcional abstrato.

Além das classes intervalares apresentadas na seção anterior, o IntLAG fornece as classes de caixas *Box* e *CudaBox* que abstraem estruturas matriciais e vetoriais. As classes possuem duas dimensões inteiras e uma única estrura *vector*[10] de elementos intervalares.

```
1 template <class T>
2 class Box{
3   public:
4     ...
5
6   private:
7     unsigned int nl, nc;
8   vector< Interval<T> > elements;
9 };
```

A classe Box e seus operadores estão definidos nos arquivos ***include/interval/box.h*** e ***include/interval/box_lib.h***. Os arquivos correspondentes à classe CudaBox são ***include/cuda_interval/cuda_box.h*** e ***include/cuda_interval/cuda_box_lib.h***.

As caixas podem representar matrizes intervalares genéricas. Os vetores são representados arbitrariamente por caixas onde a dimensão de colunas é unitária. Os construtores utilizam polimorfismo definindo vetores onde um único parâmetro inteiro é fornecido.

A classe contém diversos construtores, operações de atribuição e comparação. Estas operações estão definidas entre caixas, mutuamente, e entre caixas e *arrays* de intervalos. Alguns exemplos no código abaixo:

```
1 int main () {
2
3   Interval<double> *a, *b;
4
5   // preenche arrays
6   initialize_arrays(a, b);
7
8   Box<double> x(50, a), y(50, b), z(50, Interval<double>(0.0));
9   Box<double> A(10, 5, a), B(5, 10, a), C(10, 5);
10
11  x == a; // returna true
12    a == x; // returna true
13    x == b/ // returna false
14    x == y/ // returna false
15    A == a/ // returna true
16    A == B/ // returna false (dimensoes diferentes)
17
18    // copia os 50 primeiros elementos de a em C
19    C = a;
20    C == A // returna true
21
```

[10] Classe da biblioteca padrão.

```
22      B = A;
23      C == B // returna true
24 }
```

Além destas operações básicas, estão definidos operadores unários de sinal, assim como os operadores binários de soma, subtração e multiplicação. Os métodos BLAS apresentados na próxima sessão são utilizados na definição destes operadores.

A soma e subtração são feitas elemento a elemento. O operador de multiplicação pode significar produto por escalar (pela direita ou pela esquerda), produto de matriz-vetor ou produto de matriz-matriz a depender dos tipos e dimensões dos parêmtros. O seguinte código exemplifica estas operações:

```
1 int main () {
2
3   Interval<double> *a, *b, *c;
4
5   // preenche arrays
6   initialize_arrays(a, b, c);
7
8   Box<double> x(50, a), y(50, b), z(50, Interval<double>(0.0));
9   Box<double> A(10, 5, a), B(5, 10, b), C(10, 50, c);
10
11  // soma de vetores
12  z = x + y;
13
14  // subtracao de vetores
15  z = x - y;
16
17  // multiplicacao por escalar
18  z = 3 * z;
19
20  // multiplicacao matriz-vetor
21  z = C * a;
22
23  // multiplicacao matriz-matriz
24  A = A * B;
25 }
```

Apesar de o foco da IntLAG não estar neste aspecto, podemos ver que é possível simplificar a usabilidade das operações por meio de programação de alto nível da maneira exemplificada. Porém, para maior especialização e otimização de desempenho, o baixo nível sempre fornecerá uma alternativa melhor.

5.3 BLAS

O IntLAG não implementa tipos intervalares complexos, por isso métodos correspondentes a otimizações de matrizes hermitianas[11] comuns à pacotes BLAS são ignorados.

Também são ignorados os parâmetros de *trans* e *diag* que indicam se transposição de matrizes e matrizes triangulares unitárias devem ser consideradas, assim como incrementos.

[11] Matrizes hermitianas são iguais a suas transpostas conjugadas e portanto são unicamente determinadas pelo sua triangular superior ou inferior

Em lugar disto, a biblioteca fornece métodos para transposição matricial e uma transformação de matriz triangular em triangular unitária para que usuários possam utilizar as rotinas da forma usual, porém com menos eficiência e mais código. Estes parâmetros não representam nenhuma contribuição científica para o trabalho e imbuem o código com duplicação e complexidade

O IntLAG possui três implementações de BLAS de acordo com o paralelismo: uma serial, uma paralela com uso de OpenMP e uma para GPU usando CUDA. Os métodos seriais são especificados pelos objetos tipo Interval utilizados como parâmetros e os métodos em GPU pelos objetos CudaInterval. Os métodos que implementam paralelismo com OpenMP são contidos por um *namespace* inferior: *intlag::omp*.

Os métodos de GPU são compartimentalizados em módulos. São quatro módulos distintos que servem diferentes propósitos:

- *CudaGeneral*: este módulo contém a implementação sem uso de memória compartilhada e que não utiliza transferência de memória entre CPU e GPU. Neste caso, o gerenciamento de memória e as barreiras de sincronização devem ser explicitadas pelo programador. O uso desta módulos é interessante quando há uma sequência de operações BLAS que independem de processos de CPU, por exemplo, uma multiplicação por escalar seguida de uma multiplicação de matrizes nos mesmos dados.
- *CudaGeneralManaged*: como na CudaGeneral, este módulo não utiliza memória compartilhada, porém o sincronismo e a transferência dos dados entre a memória da CPU e GPU é implementada.
- *CudaShared*: somente os métodos de multiplicação de matriz-vetor e matriz-matriz são implementados. O módulo utiliza *kernels* com acesso a memória compartilhada. Como no CudaGeneral, é função do programador gerenciar a memória e as barreiras.
- *CudaSharedManaged*: implementa os métodos como na CudaShared, porém fazendo uso de barreiras de sincronização e gerenciamento de memória.

O uso de programação genérica elimina o uso de prefixos para os métodos BLAS[12], sendo as instâncias dos métodos definidos por um processo de parametrização.

Utilizando as técnicas explicadas até aqui, atingimos um nível de polimorfismo onde os métodos BLAS de qualquer implementação terão a mesma assinatura. Afora os incrementos e os parâmetros *trans* e *diag*, as assinaturas e especificações dos métodos seguem os mesmos padrões do Intel MKL[22].

As diferentes implementações do BLAS são disponibilizados no diretório ***include/blas*** nos seguintes arquivos:

- ***serial_blas.h*** - contém a implementação do BLAS serial utilizando a classe Interval.
- ***omp_blas.h*** - contém a implementação do BLAS com uso de OpenMP utilizando a classe Interval.

[12] No BLAS da Intel MKL, por exemplo, temos *sgemm* para multiplicação de matriz por precisão simples e *dgemm* para uso de precisão dupla.

- ***cuda_blas_kernels.h*** - contém a implementação dos kernels utilizados pelo BLAS em GPU.
- ***cuda_blas_functions.h*** - contém as interfaces do BLAS em GPU, incluindo todos os métodos que devem ser chamados pela CPU.
- ***cuda_blas.h*** - contém a implementaçãos serial dos métodos *acopy* e *swap* para a classe CudaInterval e define os métodos utilizados pelo IntLAG por padrão.

DeviceData

O gerenciamento de memória em CUDA é feito em baixo nível e portanto tende a ser deselegante. Por esta razão foi criada uma classe auxiliar *DeviceData* que torna o código mais simples e legível. O código desta classe esta disponível no arquivo ***include/aux/device_data.h*** e constam para referência no Apêndice A.

A classe compartimentaliza as operações de alocação e transferência inicial de memória no sentido host-device nos seus construtores. As operações de liberação de memória são imbutidas nos destrutores. A transferência de memória no sentido device-host é feita pelo método *toHost*.

Geometria dos Blocos

O arquivo ***include/aux/cuda_grid.h*** define de forma genérica a geometria dos blocos que é utilizada por todos os *kernels* do BLAS. O padrão para blocos unidimensionais é definido como 32 *threads* por bloco e o bidimensional é de $8x8$*threads* por bloco.

Este parâmetros afetam o desempenho dos métodos em diferentes arquiteturas e podem ser modificados apenas editando o arquivo fonte. A corretude das aplicações também pode ser impactada, a depender das arquiteturas, pois o parâmetros definem também o tamanho dos blocos de memória compartilhada que será utilizado.

Exemplo de Método em GPU

Exemplificamos aqui o código em GPU para o método *axpy* que realiza a seguinte operação:

$$axpy: \quad y = \alpha * x + y$$

Onde α é um escalar e x e y são vetores. Abaixo o código:

```
1 template <class T>
2 __global__ void ADD(CudaInterval<T> const *a, CudaInterval<T> *b, int N) {
3   int i = get_thread_id();
4   if (i < N)
5     b[i] = a[i] + b[i];
6 }
7
8 template <class T, class F>
```

```
9  __global__ void AXPY(F alpha, CudaInterval<T> const *x, CudaInterval<T> *y
       , int N) {
10   int i = get_thread_id();
11   if (i < N)
12     y[i] = alpha*x[i] + y[i];
13 }
14
15 template <class T, class F> inline __host__ static
16 void axpy(int n, F alpha, CudaInterval<T> const *x, CudaInterval<T> *y) {
17   if (alpha == 0.0) return;
18
19   DeviceData< CudaInterval<T> > d_x(n, x), d_y(n, y);
20
21   if (alpha == 1.0)
22     ADD<<<CudaGrid::blocks(n), CudaGrid::threads()>>>(x, y, n);
23   else
24     AXPY<<<CudaGrid::blocks(n), CudaGrid::threads()>>>(alpha, x, y, n)
25
26   d_y.toHost(y);
27 }
```

5.4 Tratamento de Erros

O tratamento de erros é um dos pontos fracos do IntLAG. Não foi criado um padrão por meio da criação de classes de erros específicas para a biblioteca, como é comum em bibliotecas deste tipo. Algumas fontes de erros, como alocação de memória em CPU, não são sequer tratadas.

Os erros, quando considerados, são feitos em formas diversas, com uso da função *assert*[13], uso de uma macro para tratar erros de memória em GPU (principalmente na classe *DeviceData*) e lançamento de excessões para acessos indevidos de memória nas classes de caixas. Uma particularidade do tratamento de erros é que, independente do erro encontrado, a execução é sempre interrompida.

A macro utilizada para checar erros de memória de CUDA é definida no arquivo ***include/aux/cuda_error.h***.

5.5 Testes de Validação

Todo tipo de *software* deve ser testado para assegurar a corretude da computação realizada. Isto é duplamente verdadeiro para bibliotecas algébricas.

O IntLAG foi desenvolvido majoritariamente utilizando metodologia TDD (Test Driven Development) e é, portanto, amplamente testado. Utilizamos o *framework* Google Test[15] para os testes de unidade. O IntLAG possui 153 casos testes em 4352 linhas de código, sendo diferentes métodos de atribuição, comparação e construção de uma mesma classe testados em conjunto.

A cobertura de testes é quase 100%, incluindo todos os métodos BLAS. Os únicos mé-

[13] Função da biblioteca padrão.

todos não cobertos diretamente são os arredondamentos da classe CudaRounder, que são simplórios, e alguns métodos da classe CudaInterval que são idênticos às suas contrapartes nos métodos da classe Interval, porém executam em GPU. Estes métodos são cobertos indiretamente pelos testes do BLAS.

Os testes são disponibilizados no diretório ***unit***. O diretório contém diversas suites de testes que podem ser utilizadas individualmente ou em conjunto. Alguns testes utilizam valores gerados aleatoriamente por uma classe Singleton de referência definida nos arquivos ***unit/aux/reference.h*** e ***unit/aux/reference.cu***.

Os testes de arredondamento da classe Rounder utilizam as funções *nextlarger* e *nextsmaller* definidas no arquivo ***unit/aux/ulp.h***. No arquivo ***unit/aux/test_interval.h*** são definidas macros auxiliares para testes entre intervalos, tornando desnecessário comparar explicitamente os ínfimos e supremos.

Capítulo 6

Testes de Desempenho e Resultados

O IntLAG disponibiliza um *benchmark* na pasta ***bench*** que mede os tempos de todos os métodos BLAS. Na sessão 6.4, apresentaremos os resultados obtidos nestes testes de desempenho para uma seleção restrita de métodos. Nas primeiras três sessões deste capítulo apresentaremos a metodologia, o ambiente e os parâmetros utilizados.

6.1 Metodologia

Os testes de desempenho do BLAS tiveram um papel fundamental durante todo o desenvolvimento da biblioteca, sendo utilizados cotidianamente para indicar desempenho. Uma porção significativa do tempo de projeto foi utilizada para a modelagem de um sistema de testes adequado.

Após várias iterações, desenvolvemos um modelo de testes de desempenho elegante que utiliza as estruturas de testes unitários do Google Test[15]. O modelo de testes é implementado no arquivo ***bench/aux/case.h***.

A plataforma se baseia em uma camada acima do Google Test em que definimos a classe BenchTest via herança da classe básica de testes do Google Test.

```
1 class BenchTest : public ::testing::Test {
2
3   ...
4
5   public:
6     // Test Case methods
7     void begin() {}
8     void run() {}
9     void check() {}
10    void end() {}
11    short iterations() {return 100;}
12 };
```

Neste modelo, um teste herda da classe BenchTest e implementa apenas os métodos explicitados acima, resultando em testes visualmente elegantes. O número de iterações definirá quantas vezes o método *run* será executado sequencialmente e o tempo desta sequência será medida.

Uma classe *Timer* foi implementada para tomar medidas de tempo na plataforma de

testes. A função utilizada para medir o tempo atual, que é a base da funcionalidade da classe, é a *gettimeofday* da biblioteca sys/time.h. A biblioteca reivindica uma precisão da ordem de microsegundos em processadores atuais[20].

No arquivo principal da pasta de testes, o número de vezes que cada suite de testes é executada pode ser definida e uma média é calculada. O resultado final de um execução dos testes é salva no arquivo ***bench/benchmark.txt*** e contém a semente aleatória utilizada, o tamanho das entradas e o número de repetições utilizadas para cada teste.

Disponibilizamos os testes de performance dos métodos com oito diferentes implementações do BLAS, incluindo as implementações serial e OpenMP, porém utilizando a biblioteca de intervalos do Boost[18], para comparar com aspectos alheios à biblioteca. As oito implementações são sumarizadas a seguir na ordem decrescente de desempenho esperada:

- Serial - implementação serial utilizando a classe intervalar do IntLAG.
- OMP - implementação paralela utilizando a classe intervalar do IntLAG.
- Boost - implementação serial utilizando a classe intervalar do Boost.
- Boost OMP- implementação paralela utilizando a classe intervalar do Boost.
- Cuda Float - implementação em GPU de precisão simples e uso de memória global.
- Cuda Double - implementação em GPU de precisão dupla e uso de memória global.
- Cuda Shared Float - implementação em GPU de precisão simples e uso de memória compartilhada.
- Cuda Shared Double - implementação em GPU de precisão dupla e uso de memória compartilhada.

6.2 Ambiente

Os testes de desempenho foram execuados em um PC utilizando sistema operacional Ubuntu 14.04, versão de suporte estendido. O driver Nvidia utilizado foi o *nvidia-340.29* com suporte a CUDA 6.5 e Compute Capability 3.5.

A GPU utilizada foi uma Nvidia Tesla K40c com 2880 núcleos com 0.75 GHz por núcleo. A placa possui 12GB de memória global e 1.5MB de cache L2. Por bloco, a placa disponibiliza 48KB para cache L1 e memória compartilhada, 64KB para registradores e 64KB para memória constante. O preço de mercado da Tesla K40c é U$ 3000.00.

A CPU utilizada Intel i7-4770 CPU, com 8MB de Cache, velocidade de 3.40GHz e Turbo Boost 3.9GHz. A CPU tem suporte para instruções AVX 2.0 e seu preço de mercado é U$ 300.00. A arquitetura é quadricore com *hyperthread*.

6.3 Parâmetros Utilizados

Os vetores e matrizes de entrada foram gerados aleatoriamente para cada suite de testes. Cada intervalo gerado está contido no intervalo **[-2.0, 2.0]** . O intervalo **[-2.0, 2.0]** foi

escolhido de modo a conter intervalos estritamente positivos, estritamente negativos e que contenham o zero. O valor 2 é, porém, arbitrário, pois, para os métodos testados, os valores não têm relevância.

Os intervalos dos parâmetros *alpha* e *beta* são garantidamente diferentes dos intervalos degenerados **0.0** e **1.0** para que não se alcançem condições em que parte da computação é ignorada. Para os métodos de soluções de matrizes triangulares, é garantido que os elementos da diagonal principal sejam estritamente positivos ou negativos para que os sistemas tenham solução.

O tamanho dos blocos utilizado nos casos unidimensionais foi de 64 e nos bidimensionais foi usada uma geometria 16x16. Estes valores foram ajustados empiricamente, não apresentando grande variação de resultados com seus valores vizinhos (em múltiplos de 2) e respeitam as restrições impostas no tamanho de memória compartilhada disponível na arquitetura da placa gráfica. Para os métodos que utilizam OpenMP, utilizamos oito threads, o número de núcleos de processamento virtuais da CPU.

Foram usados cinco diferentes tamanhos de entrada: $2^{17}, 2^{18}, 2^{19}, 2^{20} e 2^{21}$, onde dobrar o último valor já seria proibitivo em relação ao tempo de execução do nível 3. A dimensão dos vetores para o nível 1 variaram no conjunto dos tamanhos das entradas. Para os níveis 2 e 3, são usadas matrizes quadradas e vetores cujas dimensões são o truncamento de $\sqrt[2]{n}$ onde n é o tamanho da entrada, por simplicidade. Valores menores não foram considerados pois o interesse é o estudo assintótico dos comportamentos.

Cada suite de testes foi executada um número de 5 vezes, estas execuções são seriais, porém os testes para cada método ficam espaçados no tempo para atenuar a influência do sistema operacional. Para cada caso de teste, 100 execuções do método em questão são executadas serialmente, totalizando 500 execuções para cada método.

A versão do NVCC utilizada foi a 6.5 com GCC 4.8. Os parâmetro de compilação incluiram -O3 e -arch=sm_30, utilizando portanto o Compute Capability 3.0

6.4 Experimentos e Discussão

Escolhemos três métodos para servirem de referência, um de cada nível. Adotamos as escolhas tradicionais do BLAS, os métodos análogos: *axpy*, *gemv* e *gemm*. Estes métodos têm a seguinte forma:

$$axpy: \quad y = \alpha * x + y$$

$$gemv: \quad y = \alpha * A * x + \beta * y$$

$$gemm: \quad C = \alpha * A * B + \beta * C$$

Onde α e β representam escalares, x e y representam vetores e A, B e C representam matrizes.

A implementação do *axpy* foi apresentada no fim da sessão 5.3. As implementações do *gemv* e *gemm* são disponibilizadas apenas no código fonte da biblioteca. Para o método *axpy* não faz sentido utilizar memória compartilhada, já para os métodos *gemv* e *gemm* sim, por isso estudaremos ambas as implementações nestes casos.

6.4.1 Transferência de Memória

Antes de discutirmos os experimentos principais, mostraremos a diferença no desempenho entre os métodos de GPU que fazem a transferência de memória e os que não a fazem para os tamanhos de entrada 2^{20}, para analisarmos o custo comparado da mesma.

Os valores dos métodos sem transferência foram inferidos pela diferença entre o método correspondente e uma medida realizada somente com a transferência de memória, por comodidade da plataforma de testes utilizada.

As tabelas de valores correspondentes aos gráficos apresentados aqui e na próxima sessão estão explicitadas no Apêndice B. Apenas a média é apresentada, sem valores de desvio padrão. Qualitativamente, afirmamos que variações na segunda e terceira casas decimais foram as mais comuns. Os valores tabelados foram aproximados ao terceiro algarismo significativo.

Começamos pelo método *axpy*:

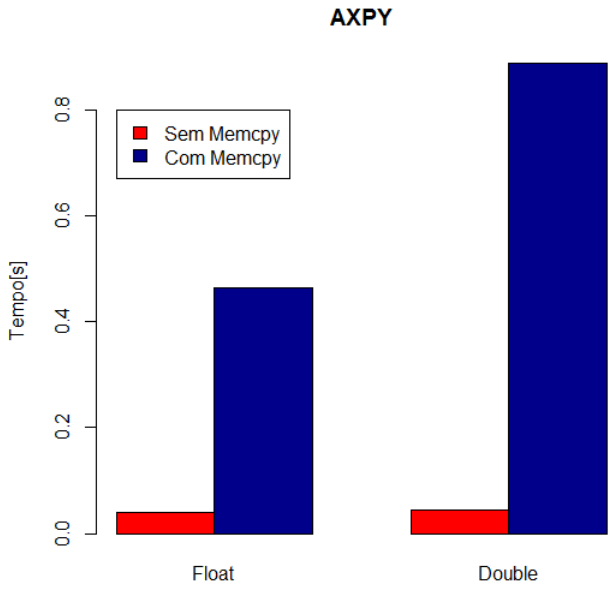

Figura 6.1: *Tempos de execução dos métodos axpy em GPU com e sem transferência de memória.*

Neste caso particular, observamos uma grande discrepância devido ao fato que a transferência de memória é linear, a mesma ordem do algoritmo utilizado. O *speedup* no caso de precisão dupla é maior que 20 e no caso de precisão simples é maior que 11.

Analisaremos agora o nível 2:

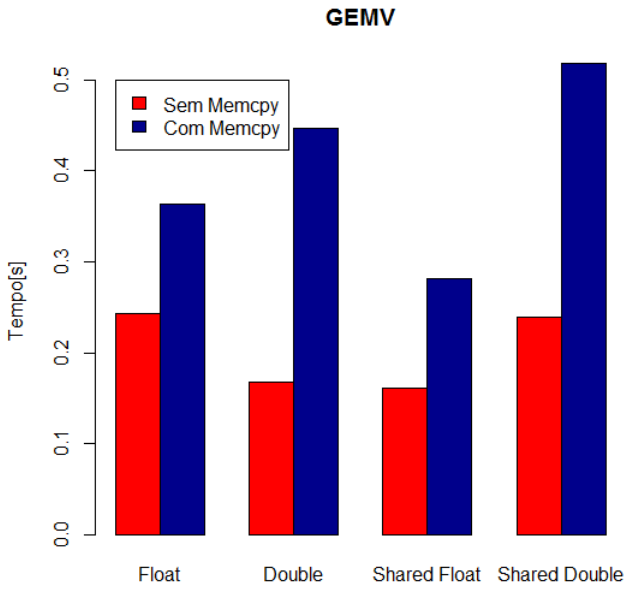

Figura 6.2: *Tempos de execução dos métodos gemv em GPU com e sem transferência de memória.*

Como esperado os *speedups* no nível dois já são mais conservadores, pois há mais computação aritmética e menos transferência de memória.

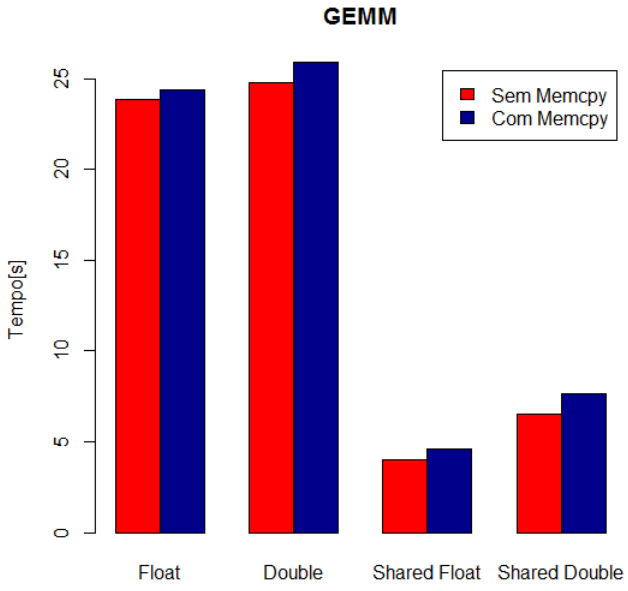

Figura 6.3: *Tempos de execução dos métodos gemm em GPU com e sem transferência de memória.*

Para os métodos de nível três, como ilustra o Gráfico 6.3, a transferência de memória tem uma influência bem reduzida, porém significativa. Estes métodos possuem uma grande razão operações de ponto flutuante por memória utilizada.

É interessante notar que em aplicações onde métodos BLAS são executados em sequência sem necessidade de controle da CPU os ganhos pela necessidade de transferir a memória apenas na primeira e última iterações se propagam.

6.4.2 Benchmark

Para os testes de desempenho dos métodos de GPU, escolhemos estudar os módulos em que há transferência de memória, pois ela faz parte de uma aplicação completa e, portanto, é mais honesto compará-los com as implementações de CPU.

Os gráficos abaixo apresentam o tempo em escala logarítmica e os tamanhos de entrada em potências de dois. Começaremos analisando o método de nível um:

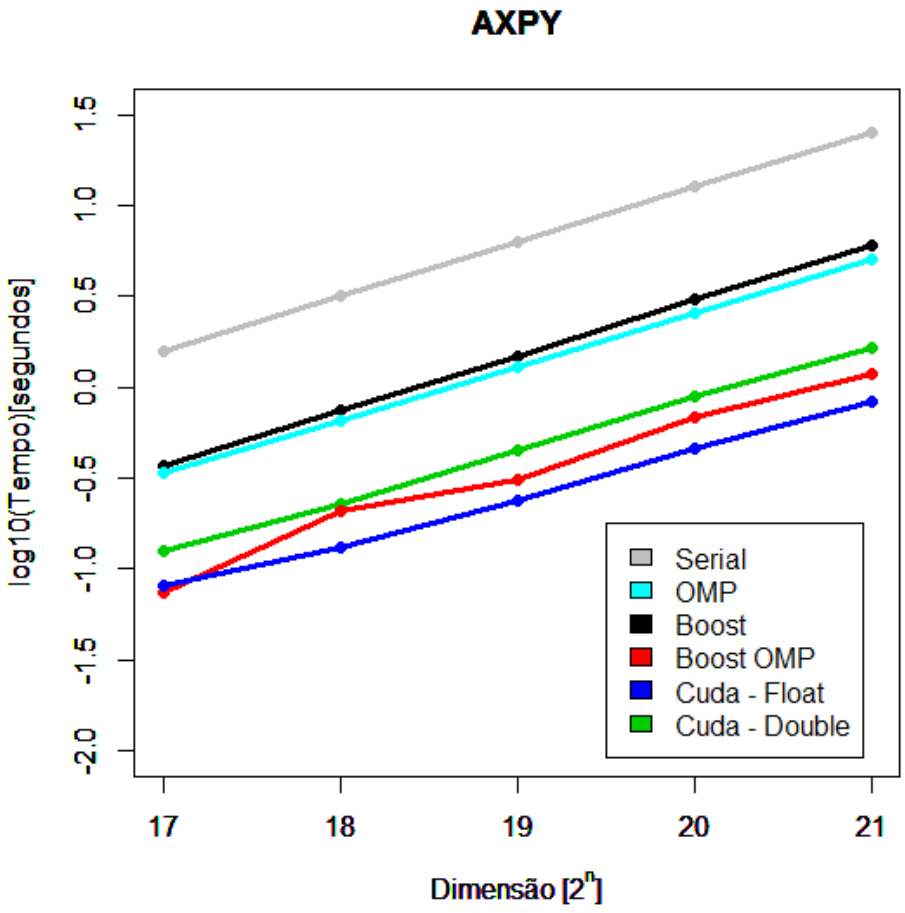

Figura 6.4: *Comparação de desempenho de métodos do tipo axpy.*

Observamos, como esperado, que a implementação serial é a pior entre as estudadas, com quase duas ordens de grandeza a mais que a melhor implementação. As implementações Boost e OMP tiveram desempenho semelhante, sendo a OMP um pouco melhor. A Boost OMP mostrou o melhor desempenho em CPU, como predito, ela chegou a ser melhor que a implementação CUDA Float para o menor tamanho utilizado, em seguida se posicionou entre a CUDA Float e CUDA Double. Também como esperado, a CUDA Double tem pior

desempenho que a implementação CUDA Float.

Uma surpresa é que as implementações em GPU tiveram um grande desempenho se comparadas com as implementações em GPU. Nos métodos de nível um, e particularmente no método *axpy* que opera em dois vetores ao invés de um, espera-se um alto custo na transferência de memória, operação que tem a mesma complexidade que o algoritmo.

O *speedup* para o tamanho máximo medido entre Boost OMP e CUDA Float foi de 1.44. Este ganho entre as implementações Serial e CUDA Float foi de mais de 30 vezes.

Abaixo estão os resultados obtidos para o método de segunda ordem, o *gemv*:

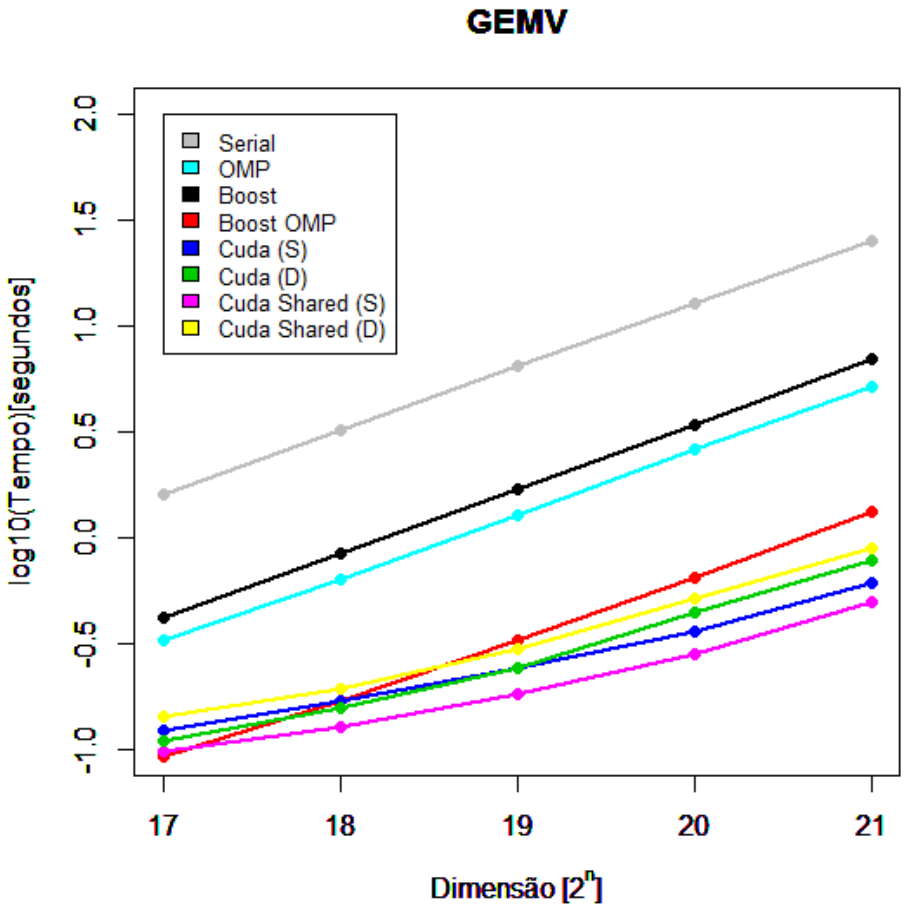

Figura 6.5: *Comparação de desempenho de métodos do tipo gemv.*

Os testes de desempenho da *gemv* indicam as mesmas observações sobre o desempenho das implementações Serial, Boost e OMP. Ainda observamos que, para o menor tamanho utilizado, o melhor desempenho é do Boost OMP, reiterando o alto custo das transferências de memória para GPU que por serem de ordem menor que o algoritmo *gemv* limitam o ganho de desempenho para pequenas dimensões.

A análise das implementações em GPU fica bastante complexa neste caso, pois há casos com uso de memória compartilhada, enquanto os outros casos fazem uso de cache L1, sendo ambos os tipos de memória de mesmo tempo de acesso. Enquanto a implementação Cuda

Shared Float obteve ganho sobre a Cuda Float, Cuda Shared Double representou uma perda em relação a Cuda Double.

Outra observação interessante é que o uso de memória compartilhada para a menor entrada utilizada foi pior em ambos os casos. Isto pode ser explicado pelo *overhead* imbutido na implementação do método que utiliza memória compartilhada.

Os *speedups* para a entrada de tamanho 2^{21} entre os métodos Serial e Boost OMP comparados com o Cuda Shared Float foram de 51 e 2.7 vezes, nesta ordem.

Finalmente, analisaremos os experimentos para o método *gemm*:

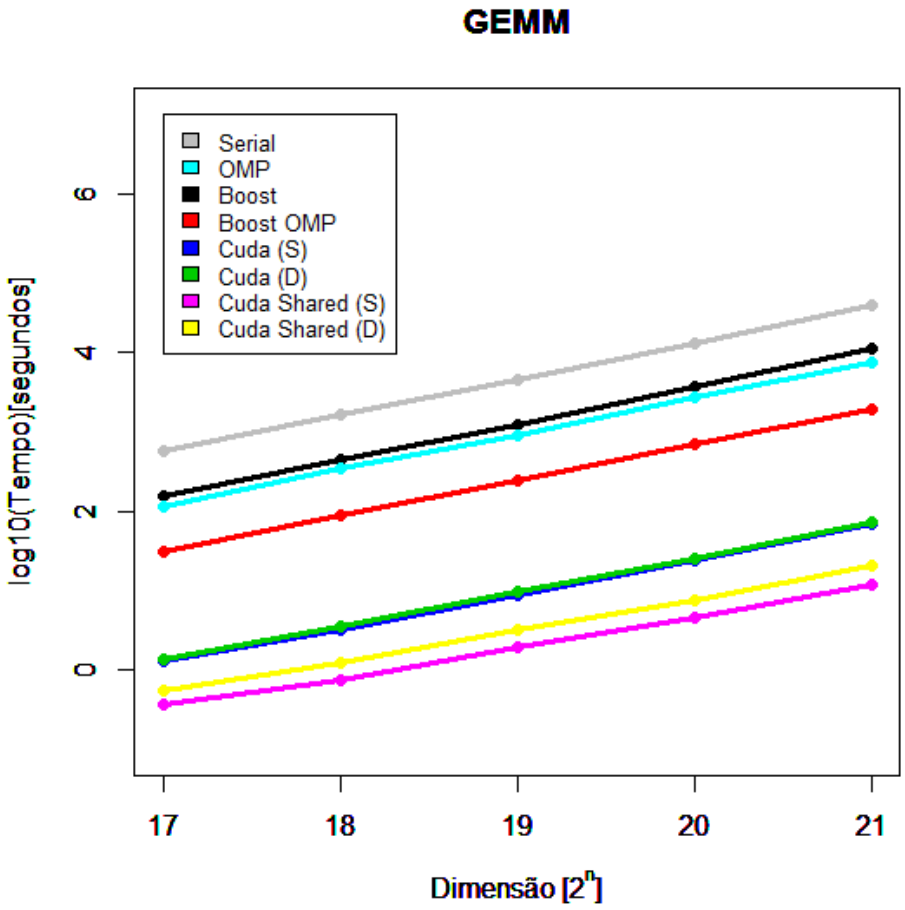

Figura 6.6: *Comparação de desempenho de métodos do tipo gemm.*

Os testes de desempenho da *gemm* não deixam muito à imaginação. Para todas as comparações de valores observados temos a relação de ordem predita anteriormente.

O nível três do BLAS é o de maior interesse, por sua maior complexidade e menor razão entre tempo de computação e transferência de memória. Aqui esperamos um grande *speedup* entre as implementações em CPU e GPU, o experimento não nos desaponta neste sentido: para a entrada de tamanho 2^{21}, a Cuda Shared Float apresenta ganhos de 3160 e 160 vezes quando comparado às implementações Serial e Boost OMP, respectivamente.

Destacamos que os algoritmos em CPU medidos aqui não são ótimos. Existem algoritmos

que utilizam poucas mudanças de modo de arredondamento, o que consideramos ser um grande gargalo destas aplicações. Finalmente, devemos ressaltar a discrepância entre o poder da placa gráfica utilizada e a CPU, simbolizada pela diferença de preços (10 vezes).

Capítulo 7

Conclusões

Construímos a biblioteca IntLAG a partir de conceitos básicos de GPGPU e aritmética intervalar. Ela implementa as funções básicas de maneira simples e legível. A biblioteca atingiu seus objetivos e consideramos que ela serve como protótipo para futuras otimizações.

A implementação de um BLAS intervalar em GPU é a maior contribuição do trabalho. Os resultados obtidos nos testes de desempenho também são animadores e sugerem futuras pesquisas de métodos intervalares em GPUs.

7.1 Sugestões para Implementações Futuras

- Eliminação da duplicação de código das classes intervalares ou fusão das duas, usando um melhor projeto de classes e programação genérica.
- Refatoração dos BLAS seriais e OpenMP para utilizarem o mínimo de trocas de modo de arredondamento: trocando o modo de arredondamento para cima e fazendo toda a computação necessária, em seguida fazer o mesmo com o modo de arredondamento para baixo e juntar os resultados da forma desejada, por último trocar o modo de arredondamento para o anterior a chamada de função.
- Estudo dos erros da biblioteca e implementação de classes de erros específicas.
- Complementação das funcionalidade do BLAS com os parâmetros de incrementos, *trans* e *diag* para conformidade.
- Atualização para novas funcionalidades de *CUDA* e *Compute Capability*.

7.2 Sugestões para Pesquisas Futuras

- Adequação da aritmética intervalar às recomendações parciais do grupo de trabalho P1788 da IEEE, incluindo divisão estendida e comparações de ordem.
- Implementação de intervalos de complexos e complemento do BLAS para matrizes hermitianas.

- Implementação de intervalos estritamente positivos e negativos para ganho de desempenho na multiplicação e divisão nestes casos.
- Implementação de intervalos pela representação centro-raio e comparações de desempenho e acurácia entre as duas representações.
- Expansão da biblioteca para inclusão de funções do LAPACK.
- Implementação de um *auto-tune* para o BLAS focando parâmetros de memória compartilhada e dimensões do *grid* de CUDA.

Apêndice A

Código Fonte

DeviceData

```
1 template <class T>
2 class DeviceData {
3
4   public:
5
6     DeviceData() : size_(0), data_(0) { }
7
8     DeviceData(int n) : size_(n*sizeof(T)) {
9       CHECKED_CALL( cudaMalloc((void**) &data_, size_) );
10    }
11
12    DeviceData(int n, const T* x) : size_(n*sizeof(T)) {
13      CHECKED_CALL( cudaMalloc((void**) &data_, size_) );
14      CHECKED_CALL( cudaMemcpy(data_, x, size_, cudaMemcpyHostToDevice) );
15    }
16
17    ~DeviceData() {
18      if (data_) CHECKED_CALL( cudaFree(data_) );
19    }
20
21    void toHost(T* x) const {
22      CHECKED_CALL( cudaMemcpy(x, data_, size_, cudaMemcpyDeviceToHost) );
23    }
24
25    void toHost(T* x, int n) const {
26      //assert( n * sizeof(T) <= size_ );
27      CHECKED_CALL( cudaMemcpy(x, data_, n*sizeof(T),
          cudaMemcpyDeviceToHost) );
28    }
29
30    T const* data() const {
31      return data_;
32    }
33
34    T* data() {
35      return data_;
36    }
37
38    DeviceData& copyAndDestroy(DeviceData& other) {
```

```
39        if (this != &other) {
40          size_ = other.size_;
41          data_ = other.data_;
42          other.data_ = NULL;
43        }
44
45        return *this;
46      }
47
48
49    private:
50      T* data_;
51      size_t size_;
52  };
53
54  template<class T>
55  void swapByReference(DeviceData<T>& a, DeviceData<T>& b){
56
57    DeviceData<T> aux;
58    aux.copyAndDestroy(a);
59    a.copyAndDestroy(b);
60    b.copyAndDestroy(aux);
```

Round Status

```
1  namespace RoundStatus {
2      const short down    = FE_DOWNWARD;
3      const short near    = FE_TONEAREST;
4      const short up      = FE_UPWARD;
5      const short chop    = FE_TOWARDZERO;
6      const short unknown  = RS_UNKNOWN;
7  }
8
9
10 inline static short getRoundMode() {
11   switch(fegetround())
12   {
13     case FE_UPWARD: return RoundStatus::up;
14     case FE_DOWNWARD: return RoundStatus::down;
15     case FE_TONEAREST: return RoundStatus::near;
16     case FE_TOWARDZERO: return RoundStatus::chop;
17     default: return RoundStatus::unknown;
18   }
19 }
20
21 inline static short setRoundMode(short rs) {
22     return fesetround(rs);
23 }
```

Apêndice B

Tabelas de Resultados

	Sem Memcpy	Com Memcpy
Float	0.039	0.465
Double	0.044	0.887

Tabela B.1: *Tabela de valores do gráfico 6.1 (axpy).*

	Sem Memcpy	Com Memcpy
Float	0.244	0.363
Double	0.167	0.446
Shared Float	0.161	0.281
Shared Double	0.239	0.518

Tabela B.2: *Tabela de valores do gráfico 6.2 (gemv).*

	Sem Memcpy	Com Memcpy
Float	23.8	24.4
Double	24.8	25.9
Shared Float	4.01	4.57
Shared Double	6.55	7.66

Tabela B.3: *Tabela de valores do gráfico 6.3 (gemm).*

	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
Serial	1.59	3.16	6.30	12.6	25.6
OMP	0.336	0.647	1.29	2.56	5.10
Boost	0.371	0.744	1.49	3.05	6.02
Boost OMP	0.0738	0.209	0.308	0.683	1.20
CUDA (S)	0.0812	0.130	0.236	0.465	0.836
CUDA (D)	0.127	0.225	0.449	0.887	1.64

Tabela B.4: *Tabela de valores do gráfico 6.4 (axpy).*

	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
Serial	1.61	3.20	6.49	12.8	25.3
OMP	0.327	0.638	1.27	2.61	5.17
Boost	0.423	0.849	1.70	3.39	6.91
Boost OMP	0.093	0.169	0.327	0.650	1.32
CUDA (S)	0.124	0.168	0.244	0.364	0.612
CUDA (D)	0.111	0.158	0.244	0.446	0.787
CUDA Shared (S)	0.0978	0.127	0.181	0.281	0.496
CUDA Shared (D)	0.144	0.194	0.300	0.518	0.893

Tabela B.5: *Tabela de valores do gráfico 6.5 (gemv).*

	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
Serial	569.3	1610.0	4570.0	13200.0	38300.0
OMP	115.0	338.0	916.0	2680.0	7570.0
Boost	153.0	440.0	1230.0	3760.0	112.0
Boost OMP	30.7	89.9	242.0	717.0	1960.0
CUDA (S)	1.26	3.23	8.90	24.4	67.8
CUDA (D)	1.40	3.54	9.53	25.9	71.5
CUDA Shared (S)	0.367	0.751	1.91	4.58	12.1
CUDA Shared (D)	0.56	1.23	3.14	7.66	20.2

Tabela B.6: *Tabela de valores do gráfico 6.6 (gemm).*

Referências Bibliográficas

[1] Mariana Kolberg, Daniel Cordeiro, Gerd Bohlender, Luiz Gustavo Fernandes, Alfredo Goldma, Multithreaded Verified Method for Solving Linear Systems in Dual-Core Processors. Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA 2008), May 2008, Trondheim, Norway. pp.9, 2008.

[2] Lawson, Hanson, et al. Basic Linear Algebra Subprograms for Fortran Usage. 1979. 25

[3] L. S. Blackford et al. An Updated Set of Basic Linear Algebra Subprograms. ACM Transactions on Mathematical Software, páginas 62-67. 2001. 25, 26

[4] Denis Deminov et al. Programming CUDA and OpenCL: A Case Study Using Modern C++ Libraries. 2013. 16

[5] László Pál e Tibor Csendes. INTLAB implementation of a global optimization method. 2009. 2

[6] Kearfott, R.B., An overview of the upcoming IEEE P-1788 working group document: Standard for interval arithmetic, IFSA World Congress and NAFIPS Annual Meeting (IFSA/NAFIPS), 2013 Joint , vol., no., pp.460,465, 24-28 June 2013. 24

[7] Edmonson, William; Melquiond, G., IEEE Interval Standard Working Group - P1788: Current Status. Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on , vol., no., pp.231,234, 8-10 June 2009. 24

[8] Tiago de Morais Montanher, Walter F. Mascarenhas. Estimação de Modelos Ocultos de Markov Usando Aritmética Intervalar. Tese apresentada ao Instituto de Matemática e Estatística da Universidade de São Paulo. Fevereiro, 2015. xv, 22, 23

[9] Ramon Moore et al. Introduction to Interval Analysis. Siam, 2008.

[10] J. M. Muller et al. Handbook of Floating-Point Arithmetic. Birkhäuser. 2010.

[11] Nicholas Wilt, The CUDA Handbook, Pearson Education Inc. 2012.

[12] NVIDIA CUDA Compute Unified Device Architecture Programming Guide, Version 1.0, 2007.

[13] Matt Pharr, editor. GPUGems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley, 2005. 19, 20, 21, 22, 31

[14] IEEE 754 Standard 14, 15
http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4610935 12, 15, 22, 23

[15] googletest xv, 10, 12, 13, 14, 15
https://code.google.com/p/googletest/wiki/Primer

[16] Wikipedia - Ray Tracing 16, 23
http://en.wikipedia.org/wiki/Ray_tracing_(graphics) 37, 39

[17] OpenMP xv, 11
http://openmp.org/wp/

[18] Boost Interval 2, 40
http://www.boost.org/doc/libs/1_35_0/libs/numeric/interval/doc/interval.htm

[19] LLNL xv, 7, 8, 9
https://computing.llnl.gov/tutorials/parallel_comp/

[20] Open Group 40
http://pubs.opengroup.org/onlinepubs/007908775/xsh/systime.h.html

[21] cuBLAS 26
https://developer.nvidia.com/cuBLAS

[22] Intel MKL 26, 35
https://software.intel.com/en-us/node/520725

[23] HPC Lab 5, 6
http://hpclab.blogspot.com.br/2011/09/is-gpu-good-for-large-vector-addition.html

[24] IntBLAS - A C++ Interval BLAS Implementation 2
http://www.oocities.org/mike_ nooner/intblas.html

[25] INTLAB - INTerval LABoratory 2
http://www.ti3.tu-harburg.de/řump/intlab/

[26] MAGMA 2
http://icl.cs.utk.edu/magma/

[27] GSL CBLAS Library 2
http://git.savannah.gnu.org/cgit/gsl.git/tree/cblas

[28] TOP500 xv, 6
http://www.top500.org/

[29] AnadTech xv, 12

[30] Wikipedia Interval Arithmetic 20
http://en.wikipedia.org/wiki/Interval_arithmetic
http://www.anandtech.com/show/7900/nvidia-updates-gpu-roadmap-unveils-pascal-architecture-f